2002

Memory optimization techniques for embedded systems

Jinpyo Hong
Louisiana State University and Agricultural and Mechanical College

MEMORY OPTIMIZATION TECHNIQUES FOR EMBEDDED SYSTEMS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Jinpyo Hong
B.E., Kyungpook National University, 1992
M.E., Kyungpook National University, 1994
August 2002

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Ramanujam for his guidance throughout this work. I would also like to thank Dr. R. Vaidyanathan, Dr. D. Carver, Dr. G. Cochdran, and Dr. S. Rai for serving on my committee, and to thank Dr. J. Trahan for his valuable advice.

I want to put some words to express my emotion, feeling and love for my mom and dad. However, after trying to do that, I gave up. I can not say thank enough with words. I just want to say this. ”MOM and DAD, I love you”. I also want to say this to my brother. ”Hi, my brother, I could come here and finish my study because I knew that you would take a good care of mom and dad. I want to thank you. I was really happy when you got married, and I was really really sorry that I couldn’t be there with you.”

I would like to express my gratitude to all my friends who made my stay at LSU a pleasant one.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

Embedded systems have become ubiquitous and as a result optimization of the design and performance of programs that run on these systems have continued to remain as significant challenges to the computer systems research community. This dissertation addresses several key problems in the optimization of programs for embedded systems which include digital signal processors as the core processor.

Chapter 2 develops an efficient and effective algorithm to construct a worm partition graph by finding a longest worm at the moment and maintaining the legality of scheduling. Proper assignment of offsets to variables in embedded DSPs plays a key role in determining the execution time and amount of program memory needed. Chapter 3 proposes a new approach of introducing a weight adjustment function and showed that its experimental results are slightly better and at least as well as the results of the previous works. Our solutions address several problems such as handling fragmented paths resulting from graph-based solutions, dealing with modify registers, and the effective utilization of multiple address registers. In addition to offset assignment, address register allocation is important for embedded DSPs. Chapter 4 develops a lower bound and an algorithm that can eliminate the explicit use of address register instructions in loops with array references.

Scheduling of computations and the associated memory requirement are closely interrelated for loop computations. In Chapter 5, we develop a general framework for studying the trade-off between scheduling and storage requirements in nested loops that access multi-dimensional arrays.

Tiling has long been used to improve the memory performance of loops. Only a sufficient condition for the legality of tiling was known previously. While it was conjectured that the sufficient condition would also become necessary for "large enough" tiles, there had been no precise characterization of what is "large enough." Chapter 6 develops a new framework for characterizing tiling by viewing tiles as points on a lattice. This also leads to the development of conditions under the legality condition for tiling is both necessary and sufficient.

CHAPTER 1

INTRODUCTION

Computer systems can be classified into two categories: general purpose systems and special purpose systems [62]. General purpose systems can be used for wide range of applications. The applications of general purpose systems are not specifically fixed [36]. Intel *86 architectures in personal computers are a typical example of general purpose systems. These kinds of systems are expected to do various jobs with reasonable performance, which means that if the application can be finished in certain amount of time, it will be considered acceptable.

As technology advances, sometimes faster than our anticipation, millions of circuits can be integrated on a single chip; this enables general purpose systems to play a great role in computing environment like workstations and personal computers. However, in some application domains, general purpose systems can not be used not only because of their performance but also due to their costs.

In some areas such as telecommunications, multimedia and consumer electronics, general purpose systems are hardly considered a competitive solution. Special purpose systems have specific application domains whose requirements of real-time performance and compact size should be achieved at any cost and even at the expense of removing some features of the systems [29]. For example, when special purpose systems to process voice signal in a cellular phone can not meet real-time performance, its output will be inaudible. Sometimes

failure of real-time performance might be even dangerous. If special purpose systems in an ABS break system of a car fail to function in real time, the result will be disastrous, but it does not mean that the situation is hopeless. The applications that will be executed on the special purpose systems are already known during the design phase of the systems, and this information is available for system designers. System designers should take advantage of this information to make the system optimized for their specific application. Digital signal processors (DSP), microcontroller units (MCU), and application-specific instruction-set processors (ASIP) are typical examples of special purpose systems.

The success of products in the market will be determined by several key factors. In case of special purpose systems, real-time performance, small size and low power consumption are the most important factors. Even if the technology advances fast, achieving high performance and low cost at the same time has been a challenging work for the system designers.

1.1 Structure of Embedded Systems

An embedded system has become a typical design methodology of a special purpose system, consisting of three main components: an embedded processor, on-chip memory, and synthesized circuit as shown in Figure 1.1. Hardware and software of an embedded system are specially designed and optimized to efficiently solve a specific problem [71]. Implementing an entire system on a single chip, so-called system-on-a-chip architecture, is profitable from the manufacturing view point [32].

Embedded systems have a strict constraint on their size because their cost heavily depends on the size [36]. Memory is the most dominant component in the size of embedded systems [10]. In order to reduce the cost, it is very crucial to minimize memory size through

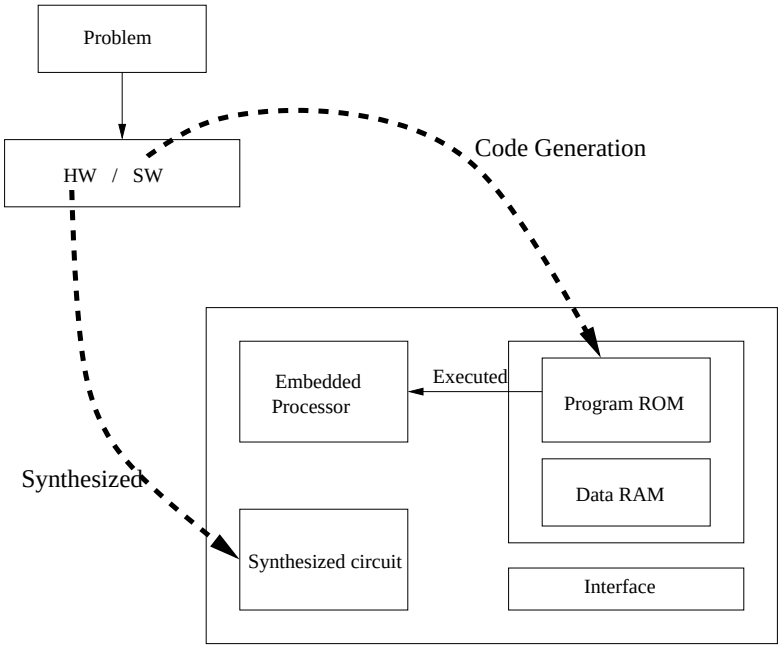


Figure 1.1: Structure of embedded systems

optimizing its usage. Memory in embedded systems consists of two parts: program-ROM and data-RAM.

Before embedded systems emerged as a design alternative of special purpose systems, there were two extreme design approaches. Figure 1.2 and 1.3 show those two approaches.

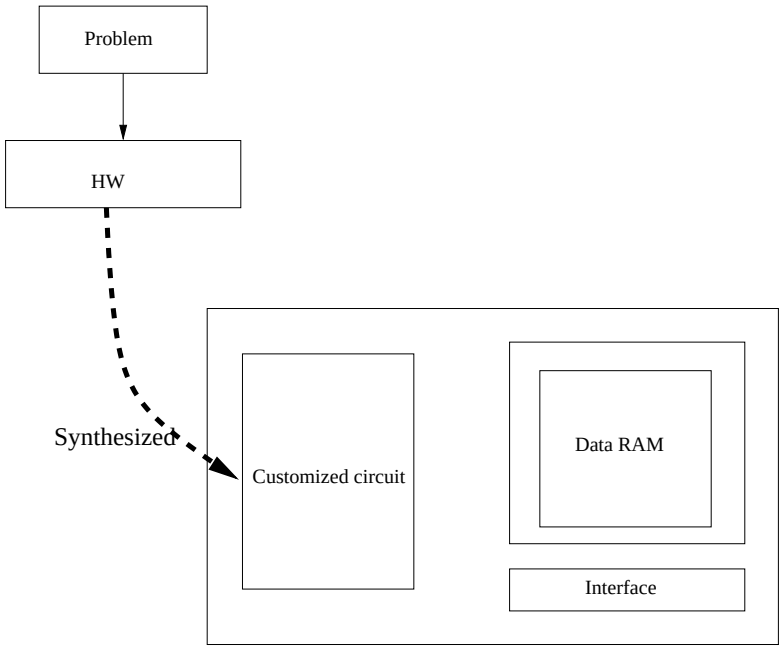


Figure 1.2: Extreme case - Only customized circuit

As it is shown in Figure 1.2, a customized circuit is synthesized for an application. The application is executed on the synthesized hardware directly. So, its real-time performance (high speed) is guaranteed, but the problem of this design is that when the application is changed for any reason, the entire system should be redesigned from the scratch because no reusable blocks exist. So, the design cost will be high. When time-to-market is crucial, this approach is a barely satisfiable solution.

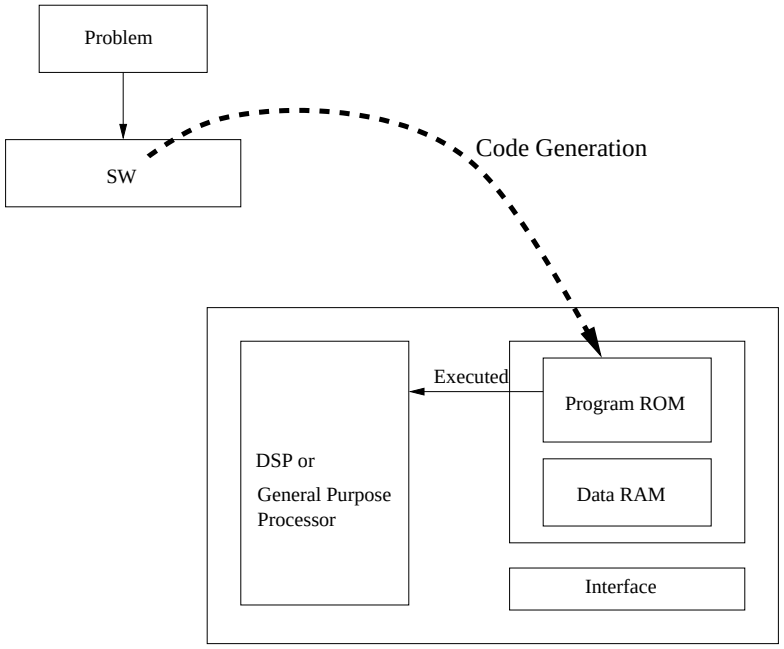


Figure 1.3: Extreme case : Only a DSP or general purpose processor

Figure 1.3 does not have a customized hardware part. In Figure 1.3, the code is generated for an application, and is burned down on the program-ROM. A DSP or a general purpose processor will execute the code. The advantage of this design is that when the application is changed, the code will be rewritten, and only the program-ROM needs to be replaced. All other components stay untouched. This approach is very adaptable to changes of the applications, but it's very difficult to achieve real-time performance and low price only with software even though these days DPSs and general purpose processors are powerful enough to tackle some specific applications like multimedia and signal processing [49]. Even though a large number of optimization techniques exist for general purpose architectures [9, 11, 26], the optimization technology of a compiler for DSPs has yet to be matured to satisfy not only real-time performance and strict requirement on the code size as well. Traditionally, a compiler for general purpose processors puts more priority on short compilation time. So, it misses aggressive optimization technology. A general purpose processor is designed to do various things with reasonable performance [36]. It may contain redundant circuits for a specific application domain, which means that the architecture of a general purpose processor is not specifically optimized for a specific application. Therefore, it's very difficult to achieve satisfiable performance with low cost by using general purpose processors. Even though a DSP , which is specialized for a specific application domain, is used in this case, it is tough to satisfy the real-time performance because the whole application will be implemented by software, and compiler optimization technology for DSPs is not matured enough.

On the contrary, in embedded systems, the application will be analyzed and then partitioned into two parts as shown in Figure 1.1 [33, 41, 75, 14, 35, 34]. One part, whose

implementation of hardware is crucial to achieve real-time performance, is to be synthesized into a customized circuit, and the other, which can be implemented by software, is to be written in high-level languages like C/C++ [42]. The critical tasks of the application will be directly executed on th synthesized circuit, and the others will be taken care of by an embedded processor. Any special purpose processor can be used as an embedded processor. Even general purpose processor can be used if it's cost-effective or imperative under certain circumstances.

1.2 Advantages of Embedded Systems

The advantages of embedded systems are as follows.

time-to-market There are many special purpose processors available for an embedded processor. Only time critical parts of an application are synthesized into a customized circuit, which reduces complexity of designing embedded systems. Using high-level languages increases the productivity of software implementation part [22].

flexibility As technology evolves, new standards emerge. For example, video coding standards evolved from JPEG [77]to MPEG1, MPEG2, and to MPEG4 [27]. This change of an application will be absorbed by rewriting software rather than re-designing an entire embedded system [76, 63]. So, embedded systems are well adaptable to application evolution. This flexibility has an effect on short time-to-market cycle and low cost [22].

real-time performance Implementation of time critical tasks in synthesized circuit helps achieve fast speed. If this goal can not be achieved, the application should be re-analyzed and re-partitioned. The optimization technology to generate code of high quality (speed) is very import to achieve this goal.

low cost Many relatively cheap special purpose processors, compared with general purpose processors, are available. Reduced design complexity by using off-the-shelve special purpose processors and synthesizing only time critical part into hardware contributes low cost of embedded systems. Generating compact code is critical to reduce cost through optimizing on-chip memory usage.

An embedded system is a superior design approach to the other two to achieve these goals, but these advantages are not automatically guaranteed just by taking an embedded system design style. In order to achieve these goals, good development tools like logic synthesis tools for hardware synthesis, a compiler for software synthesis, and a hardware-software co-simulator for hardware-software co-implementation are required [63].

1.3 Compiler Optimization for Embedded Systems

Special purpose processors that can be used as an embedded processor have different features than general purpose processors [49, 50, 48]. For example, DSPs have certain functional blocks that are specialized for typical signal-processing algorithms. A multiply-accumulation (MAC) is a typical example. DSPs can be characterized by irregular data paths and heterogeneous register files [49, 50, 47]. To reduce cost and save area, DSPs have limited data paths. With this irregular data path topology, it is not uncommon for a specific register to be dedicated to a certain function block, which means that input and output of a function unit were fixed at the time when the DSPs were designed.

Figure 1.4 shows TMS320C25 [84], one of Texas Instrument DSP series. There are three registers whose usages are specifically fixed. For example, a multiplier requires one of its operands to be from a **t** register, and its result to be stored in a **p** register. ALU's output should be stored in an accumulator. Therefore, each register should be handled differently(heterogeneousity). The data path is limited. For example, when the current

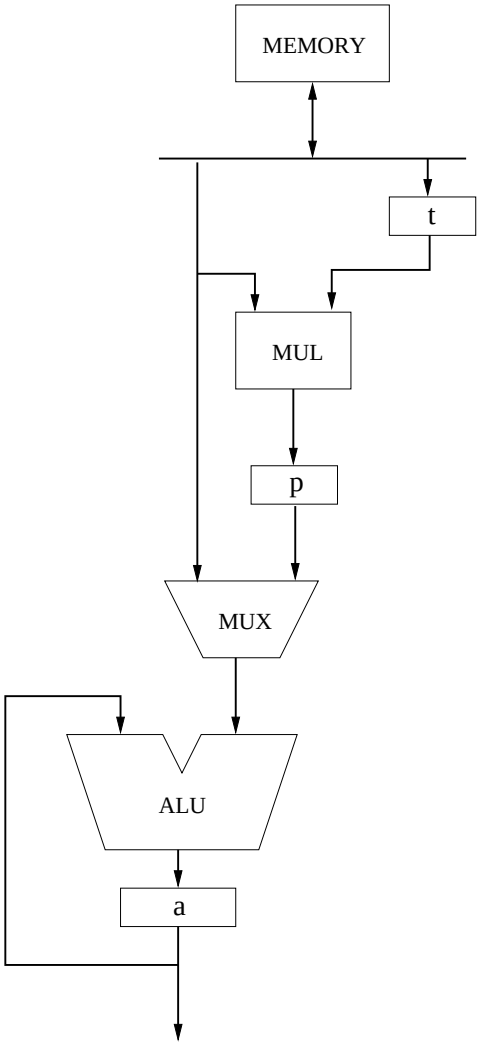


Figure 1.4: TI TMS320C25

output of ALU is needed to be input to a multiplier, the content of the accumulator can not be transfered to a multiplier directly. It should go through memory or a **t** register after going through memory (irregularity).

These structural features impose extreme difficulties on a compiler design of special purpose processors [4]. For example, heterogeneous registers cause close coupling of instruction selection and register allocation. So, when a compiler generates code, it should take care of instruction selection and register allocation at the same time [78], and also, irregular data paths affect scheduling. Therefore, an optimization technology of a compiler for special purpose processors has to take these features into account. That is the reason why optimization technology [3, 44, 60, 59, 46, 20, 61, 28] employed in a compiler of general purpose processors can not produce satisfiable results for special purpose processors.

This thesis focuses on optimization technology of a compiler for an embedded DSP processor. The generated code for an embedded DSP processor should be optimized for the real-time performance and the size at the same time.

1.4 Brief Outline

This thesis addresses several problems in the optimization of programs for embedded systems. The focus is on the generation of effective code for embedded digital signal processors and on improving memory performance of embedded systems in general.

Chapters 2, 3 and 4 address issues in generating high quality code for embedded DSPs such as the TI TMS320C25. Chapter 2 develops an algorithm to partion directed acyclic graphs into a collection of worms that can be scheduled efficiently. Our solution aims to construct the least number of worms in a worm-partition while ensuring that the worm-partition is legal. Good assignment of offsets to variables in embedded DSPs plays a key role in determining the execution time and amount of program memory needed. Chapter 3

develops new solutions for this problem that are shown to be very effective. In addition to offset assignment, address register allocation is important for embedded DSPs. In Chapter 4, we have developed an algorithm that attempts to minimize the number of address registers needed in the execution of loops that access arrays.

Scheduling of computations and the associated memory requirement are closely interrelated for loop computations. In Chapter 5, we develop a framework for studying the tradeoff between scheduling and storage requirements. Tiling has long been used to improve the memory performance of loops accessing arrays [15, 23, 80, 81, 40, 64, 65, 67, 68, 43]. A sufficient condition for the legality of tiling has been known for a while, based only on the shape of tiles. While it was conjectured by Ramanujam and Sadayappan [64, 65, 67] that the sufficient condition would also become necessary for "large enough" tiles, there had been no precise characterization of what is "large enough." Chapter 6 develops a new framework for characterizing tiling by viewing tiles as points on a lattice. This also leads to the development of conditions under the legality condition for tiling is both necessary and sufficient.

CHAPTER 2

SCHEDULING DAGS USING WORM PARTITIONS

Code generation consists in general of three phases, namely, instruction selection, scheduling and register allocation [2]. In particular, these three phases are more closely interwoven in an embedded processor system compared to a general purpose architecture because an embedded system faces more severe size, cost, performance and energy constraints that require the interactions between these three phases be studied more carefully [4].

In general, instructions of an embedded processor designate their input sources and output destinations, and instruction selection and register allocation should be done at the same time [51]. Constructing a schedule takes place after instruction selection and register allocation are done. The ordering of instructions will cause some data transfer between allocated registers and memory unit(s), and between registers and registers. As mentioned above, registers and memory have critical capacity limits in an embedded processor, which must be met. So, scheduling is very important not only because it affects the execution time of the resulting code but also because it determines the associated memory space needed to store the program.

The number of data transfers should be minimized for real-time processing and also memory capacity must be satisfied in an implementation. This chapter focuses on an efficient scheduling of control-flow directed acyclic graph (DAG) by using worm partition.

Fixed point digital signal processors such as the TI TMS320C5 are commonly used as the processor cores in many embedded system designs. Many fixed-point embedded DSP processors are accumulator-based; a study of scheduling for such machines provides a greater understanding of the difficulties in generating efficient code for such machines. We believe that the design of an efficient method to schedule the control-flow DAG is the first step in the overall task of orchestrating interactions between scheduling and memory and registers. The interactions between scheduling and registers and memory is not addressed in this chapter and is left for future work.

Aho et al. [1] showed that even for one-register machines, code generation for DAGs is NP-complete. Aho et al. [1] shows that the absence of cycles among the worms in a worm-partition of a DAG G is a sufficient condition for a legal worm-partition. Liao [51, 54] uses clauses with adjacency variables to describe the set of all legal worm-partitions and applies binate covering formulation to find optimal scheduling. He derives a set of conditions to check if a worm-partition of a DAG G is legal based on cycles in the underlying undirected graph of a directed acyclic graph G; the number of cycles in an undirected is in general exponential in the size (i.e., the number of vertices plus the number of edges) of the graph. Also, their approach to detecting a legal worm partition assumes that there are two distinct reasons that may cause a worm to be illegal, namely, (i) reconvergent paths, or (ii) interleaved sharing. Our framework shows that there is no reason to view consider these two as distinct cases. In addition, Liao [51, 54] does not provide a constructive algorithm for worm partitioning of a DAG.

The remainder of this chapter is organized as follows. In Section 2.1, we define the necessary some notation and prove the properties of graph-based structures that we define, along with a discussion of some simple examples. In addition, the necessary theoretical

framework is developed. In Section 2.2, we present and discuss our algorithm including an analysis and correctness proof based on the framework that is developed in Section 2.1. We demonstrate our algorithm by an example in Section 2.3. In Section 2.4, we present experimental results. Finally, Section 2.5 provides a summary.

2.1 Anatomy of a Worm

We begin by providing a set of definitions in connection with partitioning a DAG. Where necessary, we use standard definitions from graph theory [19]. Each vertex in the DAG under consideration corresponds to some computation. An edge represents a dependence or precedence relation between computations.

Definition 2.1 *A worm $w = (v_1, v_2, \cdots, v_k)$ in a directed acyclic graph $G(V, E)$ is a directed path of G such that the vertices, $v_i \in w, 1 \leq i \leq k, 1 \leq k \leq |V|$ are scheduled to execute consecutively.*

Definition 2.2 *A worm-partition $\mathcal{W} = \{w_1, \cdots, w_m\}$ of a directed acyclic graph $G(V, E)$ is a partitioning of the vertices V of the graph into disjoint sets $\{w_i\}$ such that each w_i is a worm.*

Figure 2.1 shows a simple example of worms. Figure 2.1-(a) is a DAG $G(V, E)$, and Figure 2.1-(b) and (c) are legal worm partitions. However, Figure 2.1-(d) shows a worm partition that is not legal, since there is no way to schedule the worms—without violating dependence constraints—such that the vertices in each worm execute consecutively. We refer to the graph whose vertices are worms and whose edges indicate dependence constraints from one worm to another (induced by collections of directed edges from a vertex in one worm to another) as a *worm partition graph.* This condition shows up as a cycle between the vertices that constitute the two worms in the worm partition graph.

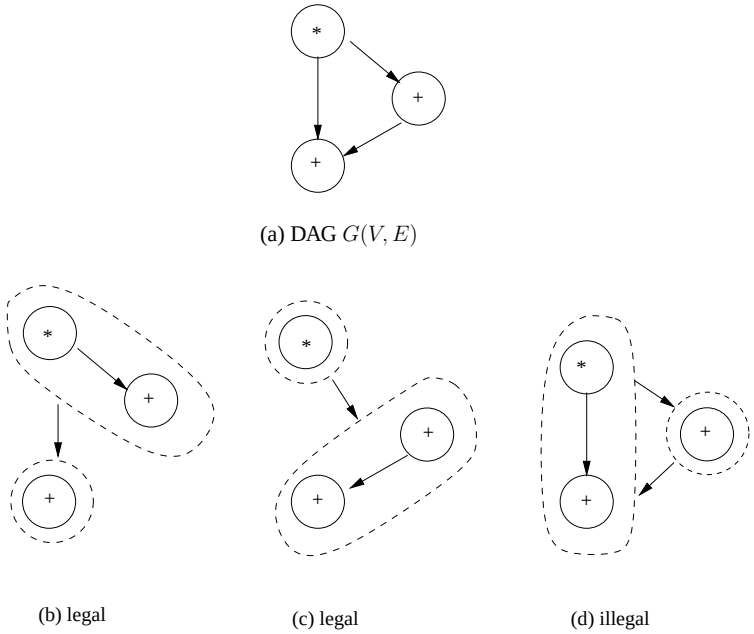


Figure 2.1: A simple example of worm partitioning.

We can assume that the DAG $G(V, E)$ is weakly connected (i.e., the underlying undirected graph of G is connected) because if a DAG $G(V, E)$ is not connected then we can schedule each disconnected component separately. For any two vertices a and b, if there are two or more distinct paths from a to b, then these paths are said to be *reconvergent;* an edge (a, b) is said to be a reconvergent edge if there is another path (this could also be another edge in the case of a multigraph) from a to b. A reconvergent edge in a worm partition graph (one that connects a vertex to itself) can cause a self-loop in a worm partition graph [51], but a self-loop does not violate the legality of a worm partition graph. Actually a self-loop in the worm partition graph (from one vertex element in a worm to a different vertex element in the same worm) is the result of a redundant dependency relation in the subject DAG. So, we can eliminate a reconvergent edge from subject DAG G without affecting the

validity of scheduling. While doing anatomy of a worm, we assume that our subject DAG G is stripped off reconvergent edges. A vertex with indegree 0 is called a *leaf.* Every vertex except the leaves in V is reachable from at least one of the leaves in V. Let V_{leaves} be the set of leaves in V.

Definition 2.3 *Let $G'(V', E')$ be an augmented graph of subject DAG graph $G = (V, E)$ such that $V' = V \cup \{S\}$ and $E' = E \cup \{(S, v_l) | v_l \in V_{leaves}\}$, where S is an additional source vertex. Each (S, v_l) is called an* s-edge.

Definition 2.4 *Let $\Psi(G, \{v\})$, $v \in V$ be a set of vertices v_t such that if there exist reconvergent paths from v to $v_t, v \neq v_t, v_t \in V$, then v_t is in $\Psi(G, \{v\})$.*

Definition 2.5 *Consider vertices u and v in a DAG $G(V, E)$. Vertex u is said to be the immediate predecessor of v if the edge $(u, v) \in E(G)$.*

Definition 2.6 *Consider vertex u in a DAG $G(V, E)$. Vertex u is said to be a predecessor of v if either $u = v$ or there is a directed path from u to v in G.*

When a vertex u has at least two different incoming edges, we have two possibilities with respect to paths to that vertex u: (a) there are two or more distinct paths (which differ at least in one vertex) from some vertex to u; or (b) there is no vertex in the graph from which there are two or more distinct paths to u. It is useful to distinguish between these two types of vertices with in-degree two or more; we introduce the notion of a *reconvergent vertex* for the former and a *shared* vertex for the latter. Note that if every vertex in a DAG is reachable from some vertex, there can not be any shared vertices in that DAG. This allows one to view every shared vertex of a DAG G as a reconvergent vertex in the corresponding augmented graph G'.

Definition 2.7 *Let v be a vertex that has indegree $k \geq 2$. Let $v_1, v_2, \cdots, v_k$ be the immediate predecessors of v. Let $P_{v_1}, P_{v_2}, \cdots, P_{v_k}$ be the set of predecessors of $v_i (1 \leq i \leq k)$. Let*

$$\mathcal{P}(v) = \bigcup_{\substack{\forall i, j, i \neq j \\ 1 \leq i, j \leq k, k \geq 2}} (P_{v_i} \cap P_{v_j}). \tag{2.1}$$

If $\mathcal{P}(v) = \phi$, then v is called a shared vertex. *Otherwise, v is called a* reconvergent vertex.

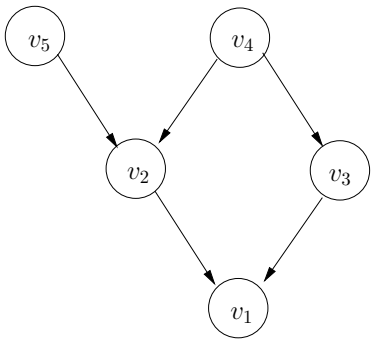


Figure 2.2: An example for Definition 2.7.

In Figure 2.2 vertices v_1 and v_2 have indegree 2. The vertex v_2 has two immediate predecessors, v_4 and v_5. The vertex v_1 has vertices v_2 and v_3 as its immediate predecessors. By Definition 2.7, $P_{v_4} = \{v_4\},\ P_{v_5} = \{v_5\},\ P_{v_2} = \{v_2, v_4, v_5\}$ and $P_{v_3} = \{v_3, v_4\}$. Then, $\mathcal{P}(v_2) = P_{v_4} \cap P_{v_5} = \{v_4\} \cap \{v_5\} = \phi$. The vertex v_2 is a shared vertex. $\mathcal{P}(v_1) = P_{v_2} \cap P_{v_3} = \{v_2, v_4, v_5\} \cap \{v_3, v_4\} = \{v_4\}$. The vertex v_1 is a reconvergent vertex. Vertices v_3, v_4, and v_5 are neither a shared vertex nor a reconvergent vertex.

Properties of Ψ

1. $\Psi(G, \{v_a, v_b\}) = \Psi(G, \{v_a\}) \cup \Psi(G, \{v_b\}),\ v_a \neq v_b,\ v_a$ and $v_b \in V$
2. $\Psi(G, V) = \bigcup_{v \in V} \Psi(G, \{v\})$
3. $\Psi(G', \{S\}) \supseteq \Psi(G, V)$
4. $\Psi(G, V_{large}) \supseteq \Psi(G, V_{small}), V_{large},\ V_{small} \subseteq V\ and\ V_{large} \supseteq V_{small}$

Proof of properties of Ψ

Proof of Property 1: If $v_t \in \Psi(G, \{v_a, v_b\})$, then v_t is to be a tail of a reconvergent path that starts from v_a or from v_b. So, v_t is to be in $\Psi(G, \{v_a\})$ or $\Psi(G, \{v_b\})$. $v_t \in \Psi(G, \{v_a\}) \cup \Psi(G, \{v_b\})$.Then, $\Psi(G, \{v_a, v_b\}) \subset \Psi(G, \{v_a\}) \cup \Psi(G, \{v_b\})$. If $v_t \in \Psi(G, \{v_a\}) \cup \Psi(G, \{v_b\})$, then v_t is a tail of a reconvergent path that starts from v_a or v_b. From the definition of Ψ, $\Psi(G, \{v_a, v_b\})$ is a set of tails of all reconvergent paths that starts from v_a or v_b. So, $v_t \in \Psi(G, \{v_a, v_b\})$. Then, $\Psi(G, \{v_a\}) \cup \Psi(G, \{v_b\}) \subset \Psi(G, \{v_a, v_b\})$.

Proof of Property 2: It is clear from Property 1.

Proof of Property 3: It is clear from the construction of G' from G that all the vertices in V are reachable from S. Without loss of generality, let v_a and v_b be the head and tail of arbitrary reconvergent paths in G from v_a to $v_b, v_a \neq v_b, v_a, v_b \in V$. Then, v_b is to be in $\Psi(G, V)$ by Property 2. Since every vertex in V is reachable from S, there is a path from S to v_a in G'. There are at least two paths from v_a to v_b which are reconvergent paths from v_a to v_b in G. There exist at least two paths from S to v_b in G'. So, v_b is to be in $\Psi(G', \{S\})$. Therefore, $\Psi(G', \{S\})$ is a superset of $\Psi(G, V)$.

Proof of Property 4: It is clear from property 2.

Theorem 2.1 *If there is a cycle C in a worm partition graph W of a subject DAG G, then there exists at least one worm in the cycle C in which there is at least one vertex with two differently oriented incoming edges.*

Proof: Without loss of generality, let this cycle C in W consist of k worms, $w_0, \cdots, w_{k-1}$ $1 < k \leq |V|$. Let the orientation of this cycle C be *lexically forward,* i.e., each edge goes from one worm to the next consecutive worm. Let $e_i,\ 0 \leq i < k$ be a lexically forward edge from a worm w_i to a worm $w_{(i+1) \bmod k}$ in the cycle C. Let $src(e_i)$ and $dest(e_i)$ be the source and destination vertices respectively of an edge e_i. Let P_{w_i} be the constituent directed path in the worm $w_i, 0 \leq i < k$. Then, P_{w_i} includes a path, p_{w_i} between $dest(e_{(i+k-1) \bmod k})$, and $src(e_i), 0 \leq i < k$ as its part. The cycle $C = e_0, p_{w_1}, e_1, p_{w_2}, \cdots, p_{w_{k-1}}, e_{k-1}, p_{w_0}$. All edges, $e_i,\ 0 \leq i < k$ have same direction because C is a directed cycle in W. Assume that all vertices in $p_{w_i}, 0 \leq i < k$ have only lexically forward edges. Then, the subject DAG G should have a directed cycle C. This contradicts the assumption that the graph G is a DAG.

Definition 2.8 *Let a vertex that has differently oriented incoming edges in C be referred to as a* bug vertex.

Lemma 2.1 *A bug vertex in G is either a shared vertex or a reconvergent vertex. There is no bug vertex that is both a shared vertex and a reconvergent vertex at the same time.*

Proof: It is clear from Definition 2.7.

Lemma 2.2 *If v is a reconvergent vertex in G, then v belongs to $\Psi(G', \{S\})$.*

(Proof) By a definition,$\mathcal{P}(v) \neq \phi$. Then, $\Psi(G, \mathcal{P}(v))$ includes v as its element and $\mathcal{P}(v) \subseteq V$. From Properties 3 and 4 of Ψ, it follows that $\Psi(G', \{S\}) \supseteq \Psi(G, V) \supseteq \Psi(G, \mathcal{P}(v))$.

Interleaved sharing may cause a cycle in W.

Lemma 2.3 *If there are shared vertices in G, then all those vertices belong to $\Psi(G', \{S\})$.*

(Proof) Any vertex v in $V(G)$ is reachable from at least one of vertices in V_{leaves} because G is a weakly connected DAG. Without loss of generality, let v_{shared} be an arbitrary shared vertex in G. Then, v_{shared} has at least two different immediate predecessors, v'_{shared} and v''_{shared}. These two predecessors of v_{shared} are reachable from some vertices v'_l and v''_l in V_{leaves}. Based on manner in which G' is constructed from G, it is clear that there are at least two paths from S to v_{shared}, one of which consists of an edge (S, v'_l), a path from v'_l to v'_{shared}, and an edge $(v'_{shared}, v_{shared})$,and the other an edge (S, v''_l), a path from v''_l to v''_{shared}, and an edge $(v''_{shared}, v_{shared})$. So, $v_{shared} \in \Psi(G', \{S\})$.

From Lemma 2.3, an augmented graph G' does not have any shared vertex because $\mathcal{P}(v)$ of a shared vertex $v \in V$ in G has at least one element S in G'.

Theorem 2.2 *If a worm w that starts from S does not include any vertices in $\Psi(G', \{S\})$, then w does not cause a cycle in a worm partition W' of G'.*

(Proof) From Lemma 2.2 and Lemma 2.3, it is clear that any augmented graph G' does not have shared vertices. From Theorem 2.1 and Lemma 2.1, the only way there can be a cycle W' is due to a reconvergent vertex, which means that it is sufficient to take care of reconvergent vertices. Assume that a worm w belongs to a cycle in W'. In order for a worm w to belong in a cycle in W', there should be at least one path P_{cycle} that goes out from w to other worm and then returns to w, which means there exist some vertex v_s and v_t in w such that v_s is an initial vertex and v_t is a terminal vertex of P_{cycle}. Any terminal vertex v_t is reachable from its predecessors in w. An initial vertex v_s is one of predecessors of v_t in w. So, we have two paths such that one of them is from S to v_t through v_s in w, and the other is from S to v_s and to v_t through the path P_{cycle}. Then, v_t should be in $\Psi(G', \{S\})$. This contradicts our assumption.

Corollary 2.1 *If a worm w satisfies a constraint $\Psi(G', \{S\})$, then it is also a legal worm in a worm partition graph W of G.*

(Proof) The only reason to introduce S is to convert potential shared vertices in G to reconvergent vertices in G'. S does not have real time step in a final scheduling. After finding a legal worm w satisfying $\Psi(G', \{S\})$, we can eliminate S from w safely without violating a legality of w. Lemma 2.3 and Property 3 of Ψ prove that this worm w is also a legal worm of a worm partition graph W of G.

Figure 2.3 shows a worm partition graph W that includes a directed worm cycle C caused by interleaved sharing [51]. In this figure, a worm $w_0 = \langle a, b\rangle, w_1 = \langle c, d\rangle, w_2 = \langle e, f\rangle$. A constituent directed path P_{w_0} is $\langle a, b\rangle$, P_{w_1} is $\langle c, d\rangle$, and P_{w_2} is $\langle e, f\rangle$. The lexically forward edges in the directed worm cycle C are $e_0 = \langle a, d\rangle, e_1 = \langle c, f\rangle$ and $e_2 = \langle e, b\rangle$; in addition, $p_{w_0} = (b, a)$ is a path between $dest(e_2)$ and $src(e_0)$, $p_{w_1} = (d, c)$ is a path between $dest(e_0)$ and $src(e_1)$, and $p_{w_2} = (f, e)$ is a path between $dest(e_1)$ and $src(e_2)$. Then, there is a cycle $C = e_0 p_{w_1} e_1 p_{w_2} e_2 p_{w_0} = \langle a, d\rangle(d, c)\langle c, f\rangle(f, e)\langle e, b\rangle(b, a)$. From Theorem 2.1, there exists a bug vertex in p_{w_0}, p_{w_1} or p_{w_2}. In this case, $\{b, d, f\}$ is the set of bug vertices. The set of immediate predecessors of the bug vertex b is $\{a, e\}$. By Definition 2.7, $P_a = \{a\}$ and $P_e = \{e\}$. Then, $\mathcal{P}(b) = \bigcup(P_a \cap P_e) = \phi$. So, the vertex b in a worm w_0 is a shared vertex. In the same way, d and f are shared vertices.

Figure 2.4 shows a worm partition graph W that includes a directed worm cycle C caused by a reconvergent vertex. In this example, W consists of 4 worms. A worm w_0 consists of a constituent directed path P_{w_0} from a vertex a to a vertex d. On the cycle C, $P_{w_0} = p_{w_1}$. In a worm w_1, P_{w_1} is from a vertex e to a vertex h, and p_{w_1} is from a vertex f to a vertex h. So, $P_{w_1} \supset p_{w_1}$. In a worm w_2, P_{w_2} is from a vertex i to a vertex m, and p_{w_2} is from a vertex l to a vertex j. So, $P_{w_2} \not\supseteq p_{w_2}$. In a worm w_3, P_{w_3} is from a vertex n to a vertex

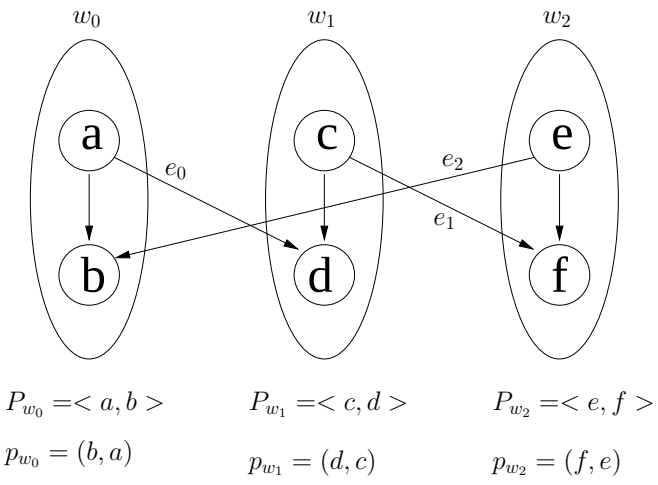


Figure 2.3: Cycle caused by interleaved sharing.

q, and p_{w_3} is from $dest(e_2)$ to a vertex p. So, $P_{w_3} \supset p_{w_3}$. Then, the directed worm cycle $C = e_0 p_{w1} e_1 p_{w_2} e_2 p_{w_3} e_3 p_{w_0}$. From Theorem 2.1, there exists a bug vertex in $p_{w_0}, p_{w_1}, p_{w_2}$, or p_{w_3}. According to Definition 2.8, differently oriented incoming edges meet in a bug vertex. It is clear that if p_{w_i} does not include a bug vertex, then $P_{w_i} \supseteq p_{w_i}$. The reason is that if there is no bug vertex in p_{w_i}, then all the edges in p_{w_i} are lexically forward and p_{w_i} can not beyond a containing worm. So, $P_{w_i} \supseteq p_{w_i}$. If a worm w_i contains a bug vertex, then $P_{w_i} \not\supseteq p_{w_1}$. According to the definition of p_{w_i}, p_{w_i} is a path between $dest(e_{(i+k-1) \bmod k})$ and $src(e_i)$. We assumed that the direction of the cycle C is lexically forward. So, all e_i's are lexically forward. If $dest(e_{(i+k-1) \bmod k})$ is an ancestor of $src(e_i)$ in a worm w_i, then p_{w_i} is a path from $dest(e_{(i+k-1) \bmod k})$ to $src(e_i)$. A p_{w_i} becomes a lexically forward directed path. Then, p_{w_i} can not have a bug vertex. So, $dest(e_{(i+k-1) \bmod k})$ can not be an ancestor of $src(e_i)$ in a worm w_i. Therefore, $P_{w_i} \not\supseteq p_{w_i}$ due to its different direction. In Figure 2.4,

$P_{w_2} \not\supseteq p_{w_2}$. So, p_{w_2} has a bug vertex that is a vertex l. A set of immediate predecessors of a bug vertex l is $\{h, k\}$. By Definition 2.7, P_h is a set of all vertices of w_0 and w_1 and vertices between a vertex n and a vertex p in w_3 and vertices i and j in w_2. P_k is a set of all vertices between a vertex i and a vertex k in a worm w_2 and between a vertex n and $dest(e_2)$ in a worm w_3. $\mathcal{P}(k) = \bigcup(P_h \cap P_k) = \{i, j\} \cup \{v | v \in$ path from n to dest$(e_2)\} \neq \phi$. So, the bug vertex l is a reconvergent vertex.

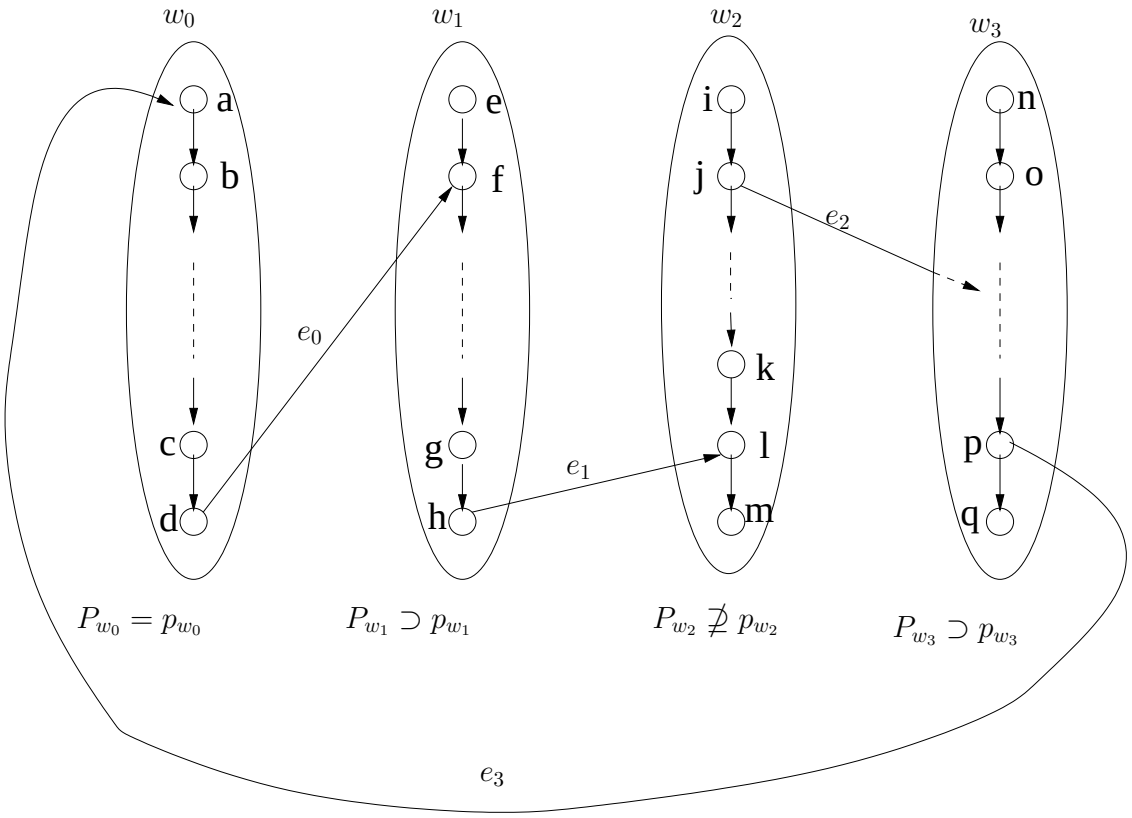


Figure 2.4: Cycle caused by reconvergent paths.

2.2 Worm Partitioning Algorithm

We use the depth-first search (DFS) [19] to find Ψ. Let us find $\Psi(G, V_{leaves})$. Choose a vertex v_l from V_{leaves}. DFS uses a stack to implement its searching such that all the vertices in a stack belong to DFS tree and every vertex in a stack is reachable in DFS tree from the bottom element (a root of DFS tree) in the stack. While applying DFS, if a non-tree edge (v_i, v_j) such as a forward edge[1] or a cross edge is visited (a back edge is impossible because G is DAG), then we know that v_j was already visited and belonged to the DFS tree. So, it is reachable from the bottom vertex in the stack (in a DFS tree), and we have another path from the bottom vertex to v_j through v_i. There exist reconvergent paths from the bottom to v_j. So, v_j should be in Ψ of the bottom vertex. Therefore, we can find Ψ by a DFS algorithm.

It is reasonably justifiable to expect that this approach may give us a better opportunity to find a longer worm by traversing a larger subtree first while constructing a DFS tree. However, it is also possible that we have an increased possibility of bug vertices in a larger subtrees. In some cases it may be useful to have information on the size of subtrees. We can get that information by traversing subtrees in postorder. To do this, first we have to get a tree of subject DAG by applying DFS or BFS, and then traverse this tree in postorder to compute the number of children of each vertex. Taking advantage of this information, we apply DFS to a subject DAG again. In our algorithm, we do not include this step because its utility depends on the particular case in hand.

Our algorithm shown in Figure 2.5 consists of several stages in which it introduces an additional source vertex S to make an augmented graph G'_i and then finds the longest legal worm that should starts from S and takes out all vertices in the legal worm from G'_i in

[1] See [19] for the classification of the edges of a graph in depth-first search.

```
1  Procedure Main
2  begin
3     G_0 ← G;
4     Construct G'_0 by introducing an additional source vertex S;
5     Eliminate reconvergent edges from G'_0;
6     i ← 0;
7     while (V_i is non-empty)
8        Find Ψ(G'_i, {S});
9        While finding Ψ(G'_i, {S}), construct DFS tree of G'_i;
10       Find the longest legal worm w_i from this DFS tree
11       by calling Find_worm(S) and Configure_worm(S);
12       G_{i+1} ← G_i − w_i,
13       where G_{i+1}(V_{i+1}, E_{i+1}), V_{i+1} = {v|v ∈ V_i ∧ v ∉ w_i}
14       and E_{i+1} = {(v_1, v_2)|v_1, v_2 ∈ V_{i+1} ∧ (v_1, v_2) ∈ E_i};
15       Construct G'_{i+1} with S;
16       i ← i + 1;
17    endwhile
18 end
```

Figure 2.5: Main worm-partitioning algorithm.

order to get a remaining subgraph G_{i+1}. In the next stage the above procedure is applied to a subgraph G_{i+1}. The reason of our introducing S successively in each stage of the algorithm is that this S prevents us from including interleaved shared vertices in worms, which was proved by Lemma 2.3. We can handle interleaved sharing in the same way as reconvergent paths. We do not need to differentiate these two cases (unlike Liao [51, 54]) in an augmented graph G'_i with S.

Assume that DFS tree is binary. In most cases, instructions in DAG have at most two operands, but this assumption is not imperative. The following algorithm can be easily adapted to higher degrees.

Correctness of the algorithm:

```
Procedure Find_worm(S)                          /* S is a pointer to vertex S */
begin
  if (S = Null)
    return −∞;
  else if (S ∈ Ψ(G'_i, {S}))
    return S.level − 1;
  else if (S is a leaf)
    return S.level;
  endif

  S.wormlength ← Find_worm(S.first_child);
                                  /* Pointer to first child of vertex S */
  S.worm ← S.first_child;
                                  /* S.worm is a pointer to a worm */
  temp ← Find_worm(S.second_child);

  if ( S.wormlength < temp )                    /* Choose a longer one */
    S.wormlength ← temp;
    S.worm ← S.second_child;
  endif

  return S.wormlength
end
```

Figure 2.6: Find the longest worm

```
Procedure Configure_worm(S)
begin
  i ← S.wormlength;
  w ← φ;
  S ← S.worm;                          /* To skip an added source vertex S */

  while (i > 0)
    w ← w ∪ {S.worm};
    S ← S.worm;
    i ← i − 1;
  endwhile

  return w;
end
```

Figure 2.7: Configure the longest worm

Let W be a worm partition graph of G. The first found worm w_0 is legal in G'_0 by Theorem 2.2, and w_0 is also legal in $G_0 = G$ by Corollary 2.1. Then, $W = \{w_0\} \cup W_1$, where W_1 is a worm partition graph of G_1. If W_1 is acyclic, then W is also acyclic. In the same way of w_0, we can find a legal worm w_1 of G_1 recursively such that $W_1 = \{w_1\} \cup W_2$. Therefore, a worm partition graph $W = \bigcup_{0 \le i \le |V|} \{w_i\}$ of G is acyclic.

Time complexity of the algorithm:

In the main procedure, Step 3 takes $O(1)$ time and Step 4 can be done in $O(|V|+|E|)$ by finding V_{leaves} and inserting the *s-edges*. The elimination of reconvergent edges can be done by finding Ψ in $O(|V| + |E|)$ and for each vertex $v \in \Psi$, by finding all common ancestors $CA(v)$ in $O(|V| + |E|)$. All the common ancestors can be found by applying $DFS(v)$ to a reverse graph G^R; G^R can be constructed in $O(|V|+|E|)$. The size of Ψ is bounded by $|V|$. If there is an edge $e =< CA(v), v >$ in G'_0, then this edge is a reconvergent edge. In this

way, we can identify all reconvergent edges. So Step 5 can be done in $O(|V|(|V|+|E|))$. The **while** loop in Lines 7–17 will iterate at most $O(|V|)$ time. In Step 8 we can find Ψ and construct a DFS tree in $O(|V|+|E|)$ time. In Step 10, Find_worm and Configure_worm can be finished in $O(|V|)$. Step 12 and Step 15 take $O(|V_i|+|E_i|)$ and $O(|V_{i+1}|+|E_{i+1}|)$ respectively. The while loop takes $O(|V|^2+|V||E|)$ time. So the proposed algorithm takes $O(|V|^2+|V||E|)$ time.

2.3 Examples

Figure 2.8 shows how our algorithm works on a DAG. In Figure 2.8-(a), vertex g is the only one leaf. An additional source vertex S is introduced and $s-edge(S,g)$ is added. $\Psi(G'_0,\{S\})$ is generated and DFS tree of G'_0 is also constructed. The longest worm $w_0=(S,g,h,i,f,c)$ is found. The edge (f,e) and (c,b) are discarded because vertex b and e are in $\Psi(G'_0,\{S\})$. Figure 2.8-(b) shows the remaining graph from which the vertices in a worm w_0 were taken out. The same procedure is repeated. A vertex S and $s-edge$ are introduced. $\Psi(G'_1,\{S\})$ is generated. DFS tree is constructed. The longest worm $w_1=(S,d,a,b)$ is found. Figure 2.8-(c) has only one vertex which is a worm w_2 by itself. Figure 2.9 shows the worm partition graph of DAG in Figure 2.8

Figure 2.10 shows an worm partition graph found by our algorithm for an example in Figure 2.3.

2.4 Experimental Results

We implemented our algorithm and applied it to several randomly generated DAGs as well graphs corresponding to several benchmark problems from the digital signal processing domain (i.e., DSPstone) [83] and from high-level synthesis [21]. Tables 2.1 and 2.2

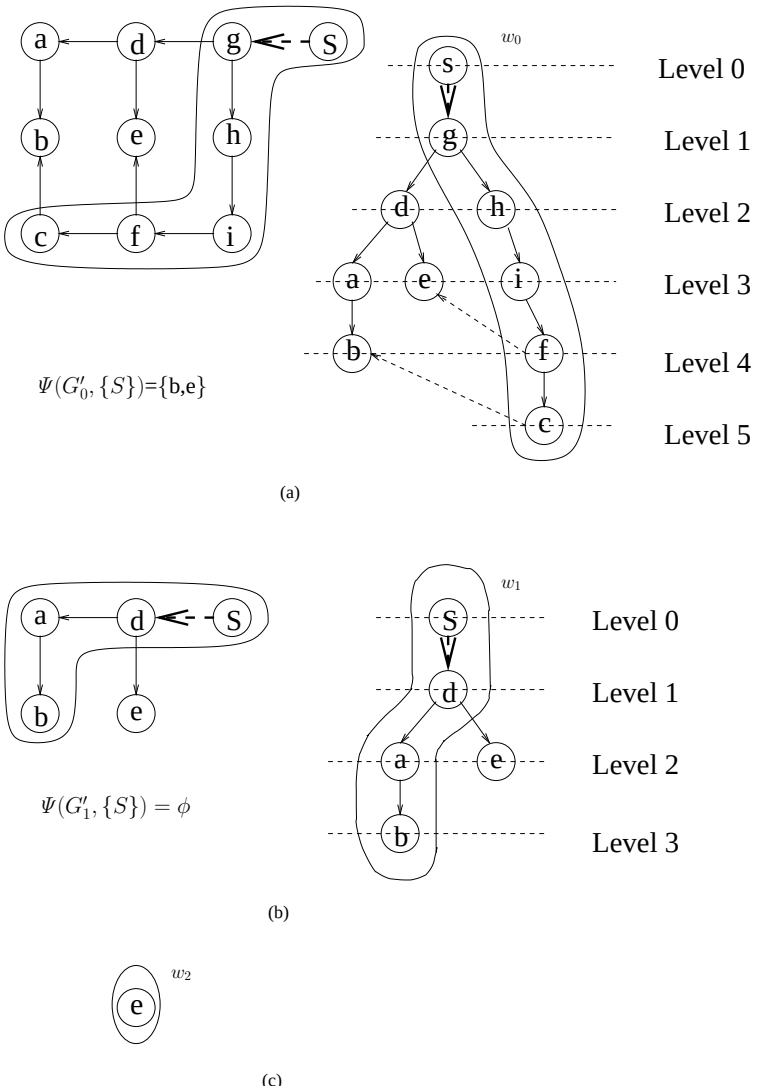


Figure 2.8: How to find a worm

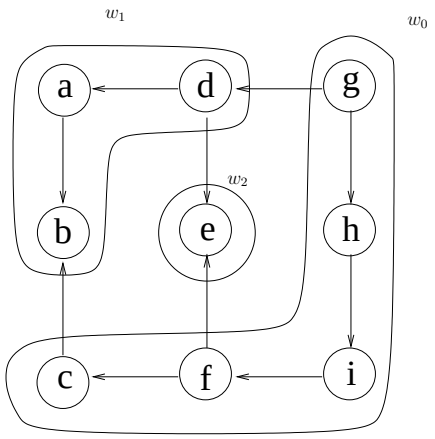


Figure 2.9: A worm partition graph

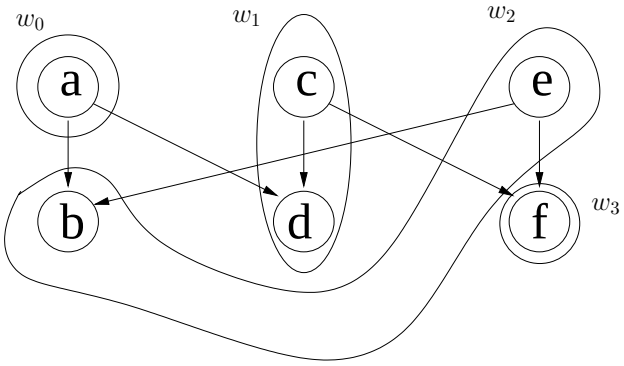


Figure 2.10: An worm partition graph for an example in Figure 2.3.

show the results on DAGs of maximum out-degree 2 and 3 respectively. Each row represents an independent experiment.

Table 2.1: The result of worm partition when max degree $= 2$

$\|V\|$	Avg.$\|W\|$	Avg. Ratio	Best Ratio	Worst Ratio
50	22.12	0.4424	0.3600	0.5400
100	44.71	0.4471	0.3600	0.5200
200	89.26	0.4463	0.3950	0.4950
300	134.20	0.4473	0.4100	0.4833
500	223.77	0.4475	0.4100	0.4880
1000	446.87	0.4469	0.4290	0.4660

In each experiment, one hundred DAGs were generated randomly. The first column is the size of the DAG, the second columns gives the average size of a worm partition graph, and the third column gives the ratio of the average size of a worm partition graph to the number of vertices in the DAG. The fourth and fifth are the ratio of lengths of the best worm partition and worst worm partition to the number of vertices of the DAG, respectively.

The result on DAGs with maximum out-degree 3 is better than the result on DAGs with maximum out-degree 2. This is because when the algorithm tries to find a longer worm, the larger out-degree DAG could give more opportunities to configure a longer worm.

We applied our algorithm to several benchmark problems. Table 2.3 shows the results. Compared with the results of randomly generated DAGs, the results on benchmark problems tend to be better. The real world problems have some kind of regularity, which can be exploited by our algorithm. In case of WDELF3, the original DAG shrunk to 6-vertex

Table 2.2: The result of worm partition when max degree $= 3$

$\|V\|$	Avg.$\|W\|$	Avg. Ratio	Best Ratio	Worst Ratio
50	21.29	0.4258	0.3400	0.5400
100	41.97	0.4197	0.3600	0.4700
200	83.49	0.4175	0.3650	0.4750
300	125.49	0.4183	0.3700	0.4600
500	210.55	0.4211	0.3940	0.4520
1000	418.97	0.4190	0.3940	0.4420

graph. The size shrunk by more than 80 percent. As an illustration, Figure 2.11 shows a worm partition graph of DIFFEQ, which is one of the benchmarks used.

Table 2.3: The result on benchmark (real) problems

Problem	$\|V\|$	$\|W\|$	Ratio($\|W\|/\|V\|$)
AR-Filter	28	12	0.4286
WDELF3	34	6	0.1765
FDCT	42	20	0.4762
DCT	48	19	0.3958
DIFFEQ	11	5	0.4545
SEHWA	32	17	0.5313
F2	22	7	0.3182
PTSENG	8	3	0.3750
DOG	11	5	0.4545

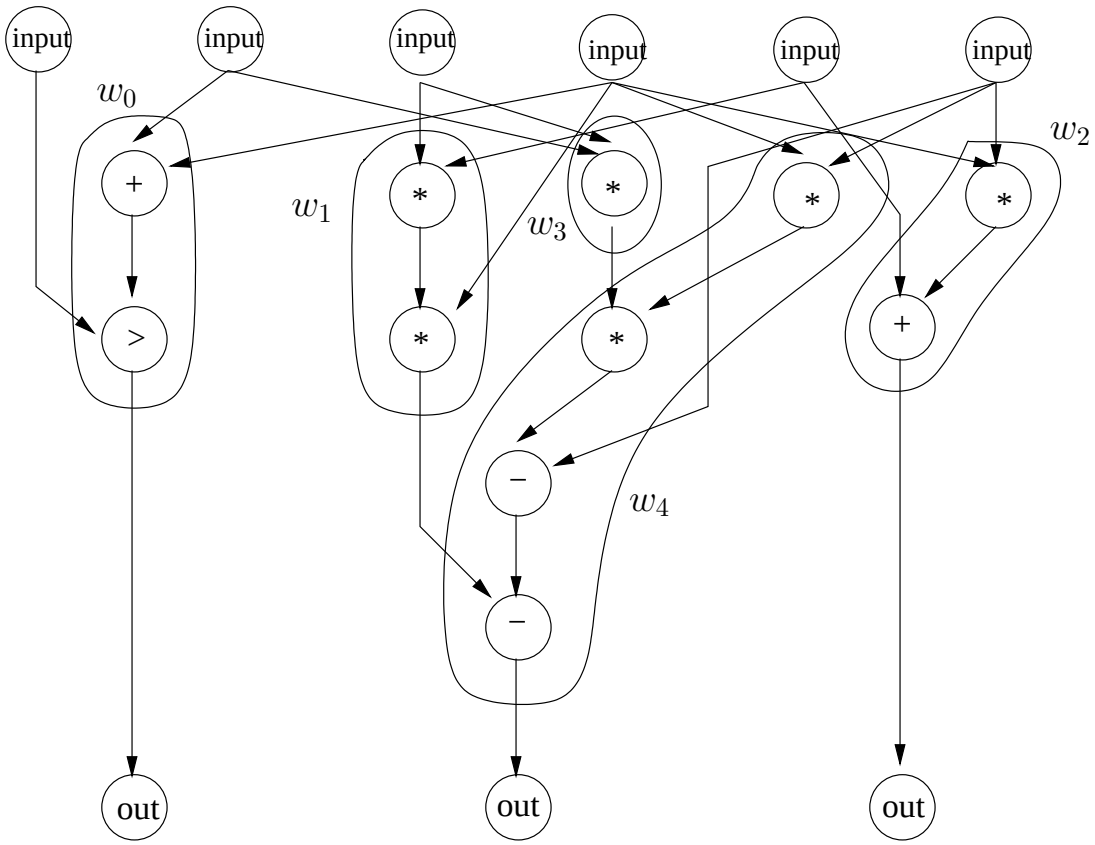


Figure 2.11: A worm partition graph of DIFFEQ

2.5 Chapter Summary

We have proposed and evaluated an algorithm to construct a worm partition graph by finding a longest worm at the moment and maintaining the legality of scheduling. Worm partitioning is very useful in code generation for embedded DSP processors. Previous work by Liao [51, 54] and Aho et al. [1] have presented expensive techniques for testing legality of schedules derived from worm partitioning. In addition, they do not present an approach to construct a legal worm partition of a DAG. Our approach is to guide the generation of legal worms while keeping the number of worms generated as small as possible. Our experimental results show that our algorithm can find most reduced worm partition graph as much as possible. By applying our algorithm to real problems, we find that it can effectively exploit the regularity of real world problems. We believe that this work has broader applicability in general scheduling problems for high-level synthesis.

CHAPTER 3

MEMORY OFFSET ASSIGNMENT FOR DSPS

With the recent shift from a pure hardware implementation to hardware/software co-implementation of embedded systems, the embedded processor has become an essential component of an embedded system. The key factor for the success of hardware/software co-implementation of an embedded system is the generation of high-quality compact code for the embedded processor. In an embedded system, the generation of a compact code should be given more priority than compilation time, which gives an embedded system designer a better chance to use more aggressive optimization techniques, and it should be achieved without losing performance (i.e., execution time).

Embedded DSP processors contain an *address generation unit* (AGU) that enables the processor to compute the address of an operand of the next instruction while executing the current instruction. An AGU has auto-increment and auto-decrement capability, which can be done in the same clock of execution of a current instruction. It is very important to take advantage of AGUs in order to generate high-quality compact code. In this chapter, we propose heuristics for for the *single offset assignment* (SOA) problem and the *general offset assignment* (GOA) problem in order to exploit AGUs effectively. The SOA problem deals with the case of a single address register in the AGU, whereas the GOA is for the case of multiple address registers. In addition, we present approaches for the case where *modify registers* are available in addition to the address registers in the AGU. Experimental

results show that our proposed methods can reduce address operation cost and in turn lead to compact code.

The storage assignment problem was first studied by Bartley [12] and Liao [51, 52, 53]. Liao showed that the offset assignment problem even for a single address register is NP-complete and proposed a heuristic that uses the *access graph,* which can be constructed for a given access sequence that involves access to variables. The access graph has one vertex per variable and edges between two vertices in the access graph indicate that the variables corresponding to the vertices are accessed consecutively; the weight of an edge is the number of times such consecutive access occurs. Liao's solution picks edges in the access graph in decreasing order of weight as long as they do not violate the assignment requirement. Liao also generalizes the storage assignment problem to include any number of address registers. Leupers and Marwedel [55] proposed a tie-breaking function to handle the same weighted edges, and a variable partitioning strategy to minimize GOA costs. They also show that the storage assignment cost can be reduced by utilizing modify registers. In [4, 5, 6, 72], the interaction between instruction selection and scheduling is considered in order to improve code size. Rao and Pande [70] apply algebraic transformations to find a better access sequence. They define the least cost access sequence problem (LCAS), and propose heuristics to solve the LCAS problem. Other work on transformations for offset assignment includes those of Atri et al. [7, 8] and Ramanujam et al. [69]. Recently, Choi and Kim [17] presented a technique that generalizes the work of Rao and Pande [70].

The remainder of this chapter is organized as follows. In Section 3.2, we propose our heuristics for SOA, SOA with modify registers, and GOA problems. We also explain the basic concepts of our approach. In Section 3.5, we present experimental results. Finally, Section 3.6 provides a summary.

3.1 Address Generation Unit (AGU)

Most embedded DSPs contain a specialized circuit called the Address Generation Unit (AGU) that consists of several address registers (AR) and modify registers (MR), which are capable of performing the address computation in parallel with data path activity. Most programs contain a large amount of addressing that requires significant execution time and space. In application-specific computing domains like digital signal processing, massive amount of data should be processed in real time. In that case, address computation takes a large fraction of execution time of a program. Due to the real time constraint faced by embedded systems, it is important to take advantage of AGUs to do address computations without consuming unnecessarily execution time; in addition, these address computations increase the size of the executed program which is detrimental to the performance of memory-limited embedded systems.

Figure 3.1 shows a typical structure of the AGU in which there are two register files, Address Register File and Modify Register File. A register in each register file will be pointed to by corresponding pointer registers, a Address Register Pointer (ARP), and a Modify Register Pointer (MRP). Usually an address register and a modify register are used as a pair, when they are employed at the same time. For example, $AR[i]$ is coupled with $MR[i]$. There are some DSP architectures where this is not the case. When the MRP contains $NULL$, the AGU will function in auto-increment/decrement mode.

Figure 3.2 shows the way the AGU computes the address of of the next operand in parallel with the data path. Figure 3.2(b) shows an initial configuration of the AGU and an accumulator in data path before the instruction, **LOAD *(AR)++** in Figure 3.2-(a) is executed. While an embedded DSP is executing the instruction, two different tasks are to be done during the same clock cycle: (i) the value stored in Loc0 pointed to by an AR is

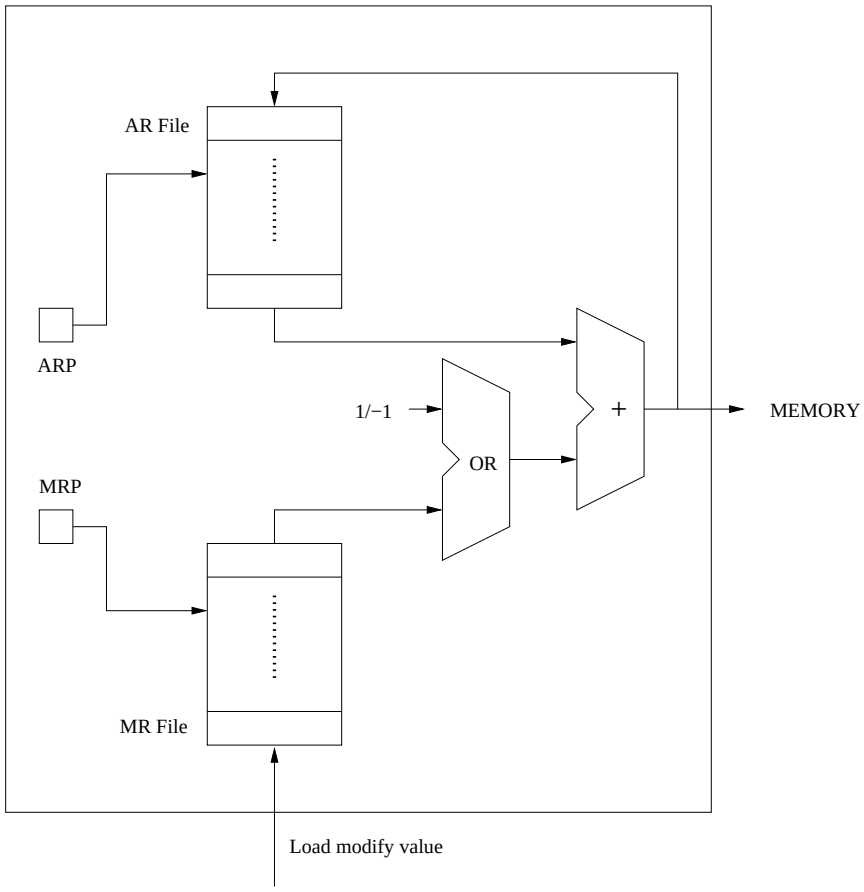


Figure 3.1: An example structure of AGU.

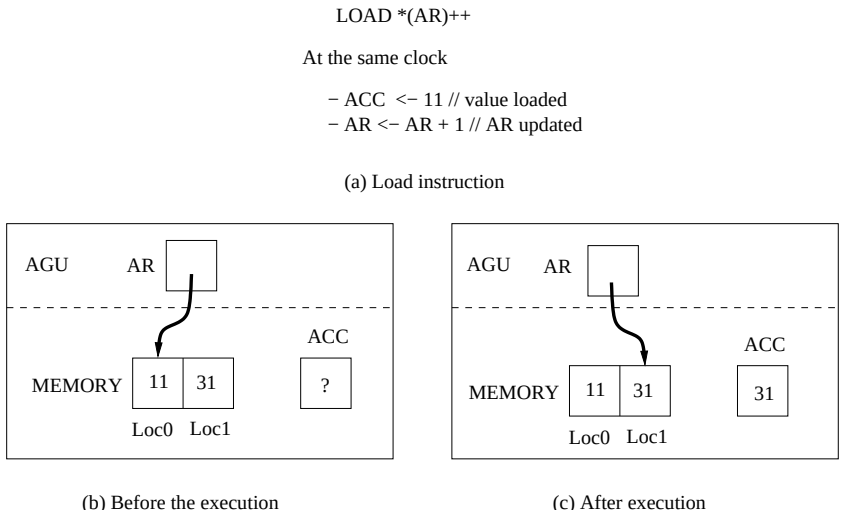


Figure 3.2: An example for AGU.

loaded into the accumulator in the data path, and (ii) the AR is updated to point an adjacent memory location, Loc1. Figure 3.2-(c) shows the configuration after the execution of the **LOAD** instruction. In this manner, two different subtasks are done in two separate circuits at the same time. From the perspective of execution time, this kind of parallel execution could be beneficial. If the value in the memory location Loc1 is an operand of the next instruction, the operand will be available immediately because the AR already points to that location.

Updating the AR to point to an adjacent memory location can be done in the AGU as shown in Figure 3.2, and also, if the offset of two memory locations is equal to the value of a modify register (MR), those two locations can be referenced shadowly by letting the AGU update an AR like $AR[i] \leftarrow AR[i] + MR[i]$.

3.2 Our Approach to the Single Offset Assignment (SOA) Problem

3.2.1 The Single Offset Assignment (SOA) Problem

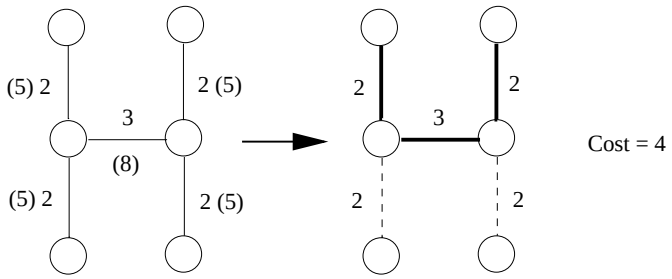
Given a variable set $V = \{v_0, v_1, \cdots, v_{n-1}\}$, the single offset assignment (SOA) problem is to find the offset of each variable $v_i, 0 \leq i \leq n-1$ so as to minimize the number of instructions needed only for memory address operations. In order to do that, it is very critical to maximize auto-increment/auto-decrement operations of an address register that can eliminate the explicit use of memory address instructions.

Liao [51] proposed a heuristic that finds a path cover of an access graph $G(V, E)$ by choosing edges in decreasing order of the number of transitions in an access sequence while avoiding cycles, but he does not say how to handle edges that have the same weight. Leupers and Marwedel [55] introduced a tie-breaking function to handle such edges. Their result is better than Liao's as expected.

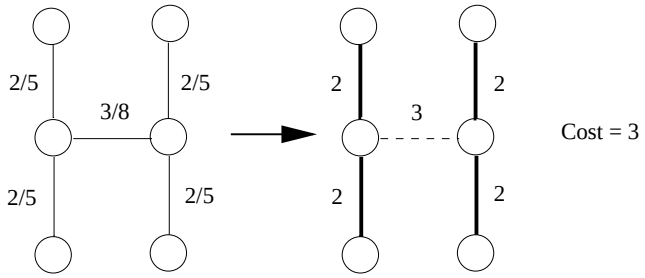
Leupers uses the sum of weights of adjacent edges as a tie-breaking function T. When two edges e_1 and e_2 have same weight, his tie-breaking function gives a higher priority to e_1 if $T(e_1) < T(e_2)$. Figure 3.3 shows how the tie-breaking function works. Figure 3.3-(a) is a given access sequence. Figure 3.3-(b) is an access graph in which each edge is assigned two values: one is the edge weight and the other (shown in parenthesis) is the value of a tie-breaking function. There are four edges with same edge weight. The edge with a weight 3 must be selected since 3 is the largest weight. In this example, a tie-breaking function will arbitrarily choose two out of the remaining edges to find a path cover because all the remaining edges have same T value. Two edges will not be selected and the resulting cost is 4. Note that the cost is the sum of the weights of the edges that have not been selected; this is exactly the same as the number of extra instructions that operate only on the address register.

An access sequence : b a b c b e d e b e f e

(a)



(b) A tie−breaking function



(c) A weight adjustment function

Figure 3.3: An example of SOA.

When an edge has a larger weight, it means that choosing that edge contributes more to reducing the cost. We may measure the preference of an edge by its weight. When an edge is selected, this selection will affect the selection of its adjacent edges in the future because in SOA, the problem is to find a path cover in which for each edge in a path cover, at most one of its adjacent edges at each of its endpoints can be selected. Selecting an edge that has a larger sum of the weights of its adjacent edges will have a greater interference impact on the cost of a path cover. We believe that the edge weight represents a preference and the sum of adjacent edges represents interference. Our weight adjustment function merges these two measurements into an adjust weight. A new weight will be given by $(\text{Preference}/\text{Interference})$. This weight adjustment function gives higher priority to edges with higher preference and less to edges with higher interference.

A new measure of weight could be a more balanced measure in the sense that it captures preference and interference at the same time. Figure 3.3-(c) shows how our weight adjustmenment function works. The preference (edge weight) of each edge is divided by its interference (the sum of the weights of adjacent edges). This example shows that a weight adjustment function may have advantage over a tie-breaking function. Our weight adjustment functions are designed to include the topology of an access graph. We propose two weight adjustment functions. Let $w(e)$ be a weight of an edge $e = (u, v)$. Let $T(e) = \sum_{(x,u)\in E} w((x, u)) + \sum_{(y,v)\in E} w((y, v))$. The first adjustment function is

$$F_1(e) = \frac{w(e)}{T(e) - 2 \times w(e)}.$$

The weight of edge e is divided by the sum of weights of its adjacent edges in $F_1(e)$. The second function is

$$F_2(e) = \frac{w(e)}{\text{The number of adjacent edges of } e}.$$

The weight of edge e is divided by the number of its adjacent edges in $F_2(e)$. We assign a new adjust weight to each edge with an adjustment function. Then, sort edges in decreasing order of the new weights, and find a path cover in the same way as Liao's. We tried another experiment in which an adjustment function F_2 is just used as a tie-breaking function. When weights (not adjust weights) of edges are same, we use F_2 as a tie-breaking function instead of using it as an adjustment function. The original weight is used as a major key and new weight returned by F_2 as a minor key during sorting.

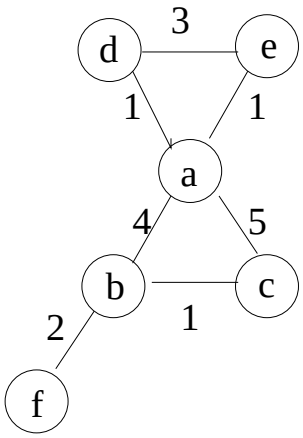
3.3 SOA with an MR register

3.3.1 A Motivating Example

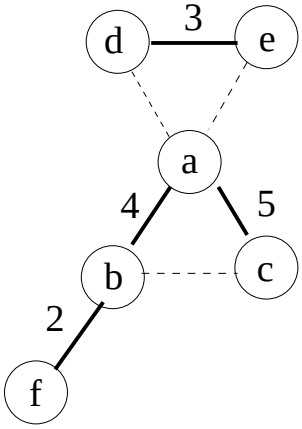
When the offset of two variables is equal to the value of a modify register (MR), those two variables can be referenced without explicit address instructions. Many DSPs include MRs in their AGUs. We observed that as edges were selected based on their weights, an access graph was fragmented into several paths. To the best of our knowledge, there have been no research on how to tackle these fragmented paths from the perspective of memory offset optimization. We believe that tackling this problem with a MR can lead to extra gains that have been missed up to now. Figure 3.4 shows our an example where this is the case.

b f b b c a c a d e d e a b a c a b a

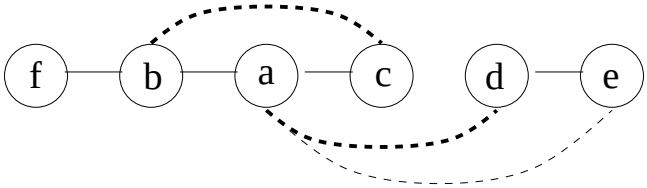
(a) an access sequence



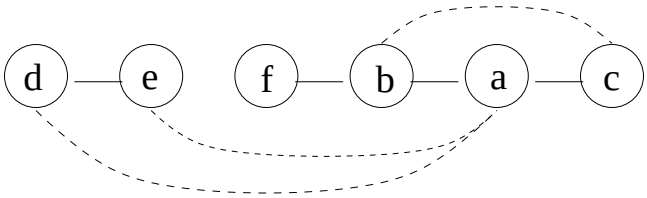
(b) an access graph



(c) fragmented paths



(d) an optimized arrangement



(e) an unoptimized arrangement

Figure 3.4: An example of fragmented paths.

In Figure 3.4-(c), two fragmented paths were generated. When the two paths are arranged in memory like in Figure 3.4-(d), two unselected edges can be recovered by assigning 2 to an MR, which means that only one unselected edge, (a, e) needs an explicit address instruction because a weight of the uncovered edge (a, e) is 1. If the two paths were to be arranged like in Figure 3.4-(e), then all three unselected edges have different offsets: 2 for (b, c), 3 for (e, a), and 4 for (d, a). Only one of them can be recovered by an MR. We propose an algorithm to handle fragmented paths.

3.3.2 Our Algorithm for SOA with an MR

Definition 3.1 *An edge $e = (v_i, v_j)$ is called an* uncovered edge *when variables that correspond to vertices v_i and v_j are not assigned adjacently in a memory.*

After applying the SOA heuristic to an access graph $G(V, E)$, we may have several paths. If there is a Hamiltonian path and SOA luckily finds it, then memory assignment is done, but we cannot expect that situation all the time. We prefer to call those paths partitions because each path is disjoint with others.

Definition 3.2 *An uncovered edge $e = (v_i, v_j)$ is called an intra-uncovered edge when variables v_i and v_j belong to the same partition. Otherwise, it is called an inter-uncovered edge. These are also referred to as intra-edge and an inter-edge respectively.*

Definition 3.3 *Each intra-edge and inter-edge contributes to an address operation cost. We call these the intra-cost and the inter-cost respectively.*

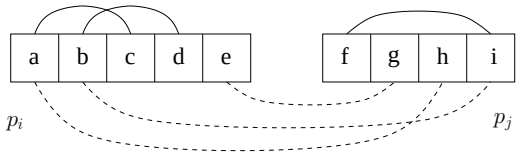
Uncovered edges account for cost if they are not subsumed by an MR register. Our goal is to maximize the number of uncovered edges that are subsumed by an MR register. The

cost can be expressed by the following cost equation.

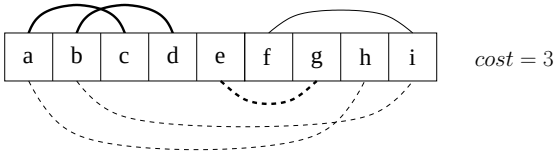
$$cost = \sum_{e_i \in intra\_edge} intra\_cost(e_i) + \sum_{e_j \in inter\_edge} inter\_cost(e_j).$$

It is very clear that a set of intra-edges and a set of inter-edges are disjoint because from Definition 3.2, an uncovered edge e cannot be an intra-edge and an inter-edge at the same time. First, we want to maximize the number of intra-edges that are subsumed by an MR register. After that, we will try to maximize the number of inter-edges that will be subsumed by an MR register. We think this approach is reasonable because when the memory assignment is fixed by a SOA heuristic, there is no flexibility of intra-edges in such a sense that we cannot rearrange them. So, we want to recover as many intra-edges as possible with an MR register first. Then, with the observation that we can change the distances of inter-edges by rearranging partitions, we will try to recover inter-edges with an MR register.

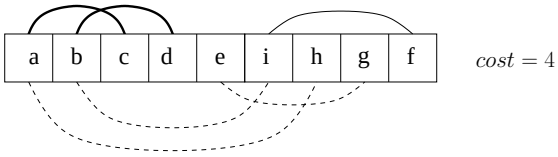
There are four possible merging combinations of two partitions. Figure 3.5 shows those four merging combinations. Intra-edges are represented by a solid line, and inter-edges by a dotted line. In Figure 3.5-(a), there are 6 uncovered edges among which there are 3 intra-edges and 3 inter-edges. So, the AR cost is 6. First, we try to find the most frequently appearing distance of intra-edges. In this example, distance 2 is the one because $distance(a, c)$ and $distance(b, d)$ are 2 and $distance(f, i)$ is 3. By assigning 2 to an MR register, we can recover two out of three intra-edges, which reduces the cost by 2. When an uncovered edge is recovered by an MR register, the corresponding line is depicted by a thick line. Next, we want to recover as many inter-edges as possible by making the distance of inter-edges 2 by applying proper merging combination. In Figure 3.5-(b), the two partitions are concatenated. One inter-edge,$e = (e, g)$ will be recovered, because $distance(e, g)$



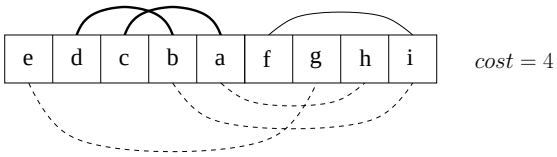
(a) $MR = 2$



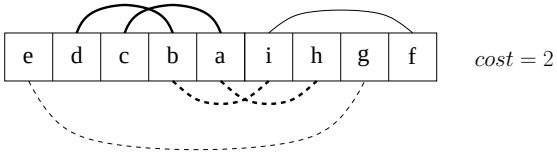
(b) $p_i \circ p_j$



(c) $p_i \circ reverse(p_j)$



(d) $reverse(p_i) \circ p_j$



(e) $reverse(p_i) \circ reverse(p_j)$

Figure 3.5: Merging combinations.

in a merged partition is 2. So, the cost is 3. In Figure 3.5-(c), the first partition is concatenated with the reversed second one. No inter-edge will be recovered. The cost is 4. In Figure 3.5-(d), the reversed first partition is concatenated with the second one. No inter-edge will be recover, either. The cost is 4. In Figure 3.5-(e), the two partitions are reversed and concatenated. It is actually equal to exchanging the two partitions. Two inter-edges will be recovered. In this case, we recover four out of six uncovered edges by applying our method. Figure 3.6 shows our MR optimization algorithm.

3.4 General Offset Assignment (GOA)

The general offset assignment problem is, given a variable set $V = \{v_0, v_1, \cdots, v_{n-1}\}$ and an AGU that has k ARs, $k > 1$, to find a partition set $\mathcal{P} = \{p_0, p_1, \cdots, p_{l-1}\}$, where $p_i \cap p_j = \phi, i \neq j, 0 \leq i, j \leq l-1$, subject to minimize GOA cost

$$\sum_{i=0}^{l-1} SOA\_cost(p_i) + l,$$

where l is the number of partitions, $l \leq k$. The second term l is the initialization cost of l ARs. Our GOA heuristic consists of two phases. In the first phase, we sort variables in descending order of their appearance frequencies in an access sequence, i.e., the number of accesses to a particular variable. Then, we construct a partition set $\mathcal{P}$ by selecting the two most frequently appearing variables, which will reduce the length of the remaining access sequence most, and making them a partition, $p_i, 0 \leq i \leq l-1$.

After the first phase, the way we construct a partition set $\mathcal{P}$, we will have $l, l \leq k$, partitions that consist of only 2 variables each. Those partitions have zero SOA cost, and we have the shortest access sequence that consists of $(|V| - 2l)$ variables. In the second phase, we pick a variable v from the remaining variables in the descending order of frequency, and

Procedure SOA_mr
begin
$G_{partition}(V_{par}, E_{par})$ $\leftarrow$ Apply SOA to $G(V, E)$;
$\Phi_{m\_sorted} \leftarrow$ sort m values of edges (v_1, v_2) by frequency in descending order;
$M \leftarrow$ the first m of Φ_{m_sorted};
$optimizedSOA \leftarrow \phi$;

for each partition pair of p_i and p_j **do**
Find the number, $m_{(p_i,p_j)}$ of edges, $e = (v_1, v_2)$, $e \in E$, $v_1 \in p_i$, $v_2 \in p_j$
such that their distance (m value) $= M$ from four possible merging combinations,
and assign a rule number that can generate $m = M$ most frequently to (p_i, p_j);
enddo

$\Psi_{sorted\_par\_pair} \leftarrow$ Sort partition pairs (p_i, p_j) by $m_{(p_i,p_j)}$ in descending order;

while ($\Psi_{sorted\_par\_pair} \neq \phi$) **do**
$(p_i, p_j) \leftarrow$ choose the first pair from $\Psi_{sorted\_par\_pair}$;
$\Psi_{sorted\_par\_pair} \leftarrow \Psi_{sorted\_par\_pair} - \{(p_i, p_j)\}$;
if ($p_i \notin optimizedSOA$ and $p_j \notin optimizedSOA$)
$optimizedSOA \leftarrow (optimizedSOA \circ merge\_by\_rule(p_i, p_j))$;
$V_{par} \leftarrow V_{par} - \{p_i, p_j\}$;
endif
enddo

while ($V_{par} \neq \phi$) **do**
Choose p from V_{par};
$V_{par} \leftarrow V_{par} - \{p\}$;
$optimizedSOA \leftarrow (optimizedSOA \circ p)$;
enddo

return $optimizedSOA$;
end

Figure 3.6: Heuristic for SOA with MR.

choose a partition p_i such that $SOA\_cost(p_i \cup \{v\})$ is increased minimally, which means that merging a variable v into that partition increases the GOA cost minimally. This process will be repeated $(|V| - 2l)$ times, till every variable is assigned to some partition.

Figure 3.7 shows our GOA algorithm that consists of two while loops. The first while loop implements the first phase and the second the second phase. We need to sort variables. Let L be a length of an access sequence. It takes $O(|V|log|V| + L)$ time. We also need to solve SOA of the entire variables in order to use that SOA cost as an initial best cost at the beginning of the first phase in which the cost will be used to decide whether a further partitioning continues or not. It takes $O(|E|log|E|)$ time. The first while loop iterates at most k times. In each iteration, SOA is to be solved with remaining variables to compute the sum of GOA cost of partitions and SOA cost of the remaining variables. It takes $O(|E|log|E|)$ time. So, the first while loop takes $O(k|E|log|E|)$ time. The second loop iterates $(|V| - 2l)$ times. In each iteration, l SOA problems need to be solved, where $l \leq k$. It takes $O(l|E|log|E|)$ time. So, the second one takes $O(l(|V| - 2l)(|E|log|E|))$ time. The time complexity of our GOA_FRQ is $O(k(|V| - 2k + 1)(|E|log|E|) + |V|log|V| + L)$.

3.5 Experimental Results

We generated access sequences randomly and apply our heuristics, Leupers' and Liao's. We repeated the simulation 1000 times on several problem sizes. Table 3.1 shows the results of several SOA heuristics. The first column shows a problem size. The second column shows AGU configurations on which we experiment several heuristics. There is only one AR in a coarse configuration. The W_mr row represents a 1-AR and 1-MR AGU. The third row, W_mr_op, has the same AGU configuration as W_mr, but we apply our optimization heuristic of rearranging and merging path partitions to recover uncovered edges with an MR register. The third and fourth columns are results of Liao's and of Leupers' heuristics,

Procedure GOA_FRQ(V, s, k)
V : a set of variables
s : an access sequence
k : the number of ARs
begin
  $V_{sort} \leftarrow$ Sort variables in the descending order of frequency in s;
  $i \leftarrow 0$;
  $best\_cost \leftarrow SOA\_cost(V, s) + 1$;

  while ($i < k$ **and** $|V_{sort}| > 1$) **do**
    pick the first two variables v_a and v_b from V_{sort};
    $V_{sort} \leftarrow V_{sort} - \{v_a, v_b\}$;
    $V_i \leftarrow \{v_a, v_b\}$;
    $new\_cost \leftarrow (SOA\_cost(V_{sort}) + 1) + (i + 1)$;
    if ($new\_cost \leq best\_cost$)
      $best\_cost \leftarrow new\_cost$;
      $i \leftarrow i + 1$;
    else
      $i \leftarrow i + 1$;
      break ;
    endif
  enddo

  $l \leftarrow i$;

  while ($V_{sort} > 0$) **do**
    $v \leftarrow$ pick a first variable from V_{sort};
    $V_{sort} \leftarrow V_{sort} - \{v\}$;
    for $j \leftarrow 0$ **to** $l - 1$ **do**
      $cost_j \leftarrow SOA\_cost(V_j \cup \{v\})$;
    enddo
    $index \leftarrow$ find minimum cost partition;
    $V_{index} \leftarrow V_{index} \cup \{v\}$;
  enddo

  return $(V_0, V_1, \cdots, V_{l-1})$;
end

Figure 3.7: GOA Heuristic.

and the remaining columns show the results of ours. The results in a coarse row of Table 3.1 do not include an initialization cost of a AR. Usually, the SOA cost does not include initialization cost of an AR (but not necessarily). So, for a fair comparison with the result of a coarse configuration, the results of W_mr and W_mr_op do not include the initialization cost of a AR, either. However, the initialization cost of an MR is included.

For all experiments of different problem sizes, the results from Leupers' and ours are better than Liao's in a coarse AGU configuration. It is very difficult to pick the best one among Leupers' and ours. That is the reason why we iterate this simulation 1000 times. Among those nine experiments, in only one case the performance of Leupers' is the best in case of $|S| = 100, |V| = 80$. Even in that case, it is tied with our heuristic F_1. In other eight experiments, our heuristics are slightly better than Leupers'. The results of W_mr prove that introducing an MR register in AGU can significantly improve the performance of AGU. There is an interesting trend in W_mr result. In three experiments of $|S| = 10, |V| = 5$, $|S| = 20, |V| = 5$, and $|S| = 100, |V| = 10$, Liao's heuristic is better than the others. We feel that the experiment of $|S| = 10, |V| = 5$ is too small to say some trend. The common fact of the other two cases is that the percentage of the number of variables in an access sequence to the length of an access sequence is relatively low (below 25 percent).

The result of W_mr_op shows that applying our MR heuristic to recover uncovered edges is crucial to enhance the performance of AGU by exploiting an MR aggressively. Our MR optimization heuristic reduces the costs of every experiment of every heuristic.

We experimented another interesting simulation in which we introduce a kind of tie-breaking function for F_1 and F_2. In other words, after a new weight is assigned to all edges with our adjustment function, a tie-breaking function is enforced. However, we observe no

gains at all. We think it is due to the fact that there are very few chances for edges to have a same weight because many of new weights are not integer.

Table 3.2 and 3.3 show the results of Leupers' GOA and of our GOA_FRQ heuristic. We iterate simulation 500 times. Leupers' GOA algorithm uses his SOA algorithm as its SOA subroutine. Our GOA_FRQ uses F_1 as its SOA subroutine. The first column shows AGU configurations. The second and third columns are results of Leupers' and of our GOA_FRQ respectively. We include 1AR_1MR and ARmr_op results. On the contrary to the Table 3.1, these results include an initialization cost of an AR in order to be fairly compared with the results of GOA heuristics on a 2-AR AGU.

Except for some rare anomalous cases such as a 6-AR AGU of $|S| = 50, |V| = 25$, a 8-AR AGU of $|S| = 100, |V| = 50$, and a 10-AR AGU of $|S| = 100, |V| = 50$, our GOA_FRQ is better than Leupers'. We think the reason is that from the way that GOA_FRQ takes out the most frequently appeared two variables and assigns them to an AR register, the shorter length of the remaining access sequence could contribute to our GOA_FRQ's better performance.

Table 3.1 already showed that introducing an MR can improve the AGU performance and that an optimization heuristic for an MR register is needed to maximize a performance gain. Table 3.2 and 3.3 show that the results of 2-AR AGU are alway better than 1AR_1MR's and even ARmr_op's. It is because even if we apply a MR optimization heuristic, which is naturally to be more conservative than GOA heuristic of 2-AR in such a sense that only after several path partitions are generated by SOA heuristic on entire variables, a MR optimization heuristic would try to recover uncovered edges whose occurrences heavily depend on SOA heuristic, a GOA heuristic can exploit a better chance by partitioning variables into two sets and applying SOA heuristic on each partitioned set.

However, GOA's gain over ARmr_op does not come for free. The cost of the partitioning of variables might not be negligible as it was shown in section 3.4. However, from the perspective of performance of an embedded system, our experiment shows that it is better to pay that cost to get performance gain of AGU. The gain of 2-AR GOA over ARmr_op is noticeable enough to justify our opinion.

Our GOA results show that when the problem size is fixed, it may not be beneficial to introduce too many address registers. Beyond a certain point of threshold, introducing more ARs may not be beneficial and sometimes even be harmful. For example, when a problem size is $|S| = 50, |V| = 25$, we observe such a lose of a gain between 7-AR and 8-AR configurations. There are other such phenomena between 5-AR and 6-AR of $|S| = 50, |V| = 40$, and between 7-AR and 8-AR for Leupers' and 8-AR and 9-AR for ours in case of $|S| = 100, |V| = 80$.

When an AGU has several pairs of a AR and an MR, in which AR[i] is coupled with MR[i], our path partition optimization heuristic can be used for each partitioned variable set. Then, the result of each pair of the AGU will be improved as we observed in Table 3.1.

Figures 3.8, 3.9, 3.10, 3.11 show bar graphs based on the results in Table 3.1. When an access graph is dense, all five heuristics perform similarly as shown in Figure 3.8. In this case, introducing a mr optimization technique does not improve performance much. Figure 3.9, 3.10 show that when the number of variables is 50% of th length of an access sequence, introducing optimization technique can reduce the costs. Figure 3.10 shows that when the access graph becomes sparse, the amount of improvement becomes smaller than when the graph is dense, but it is still reduce the costs noticeably. Except the case when an access graph is very dense like in Figure 3.8, applying our mr optimization technique is beneficial in all heuristics including Liao's and Leupers'.

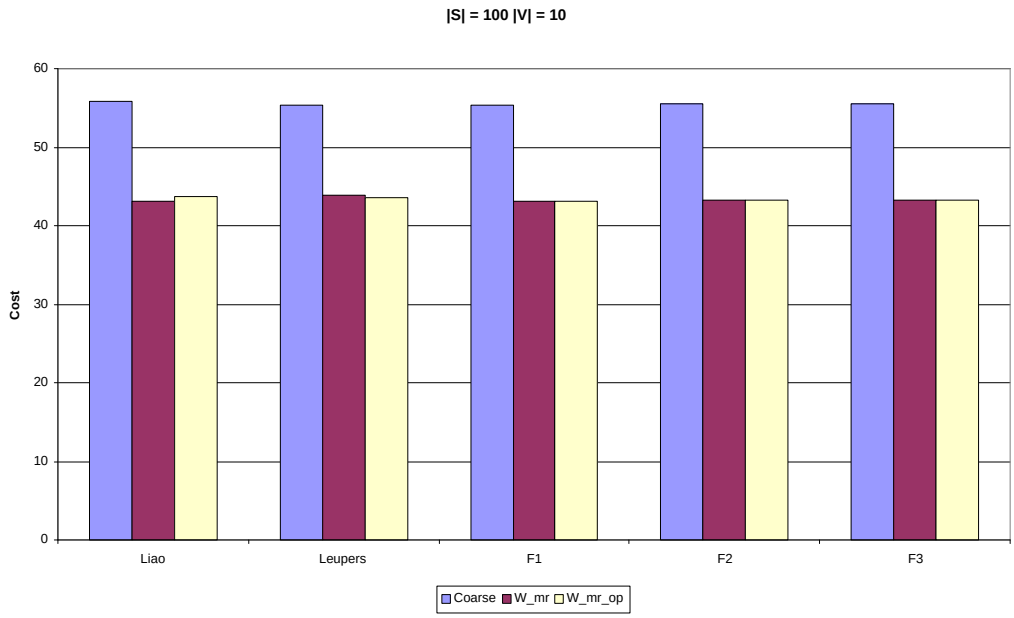


Figure 3.8: Results for SOA and SOA_mr with $|S| = 100, |V| = 10$.

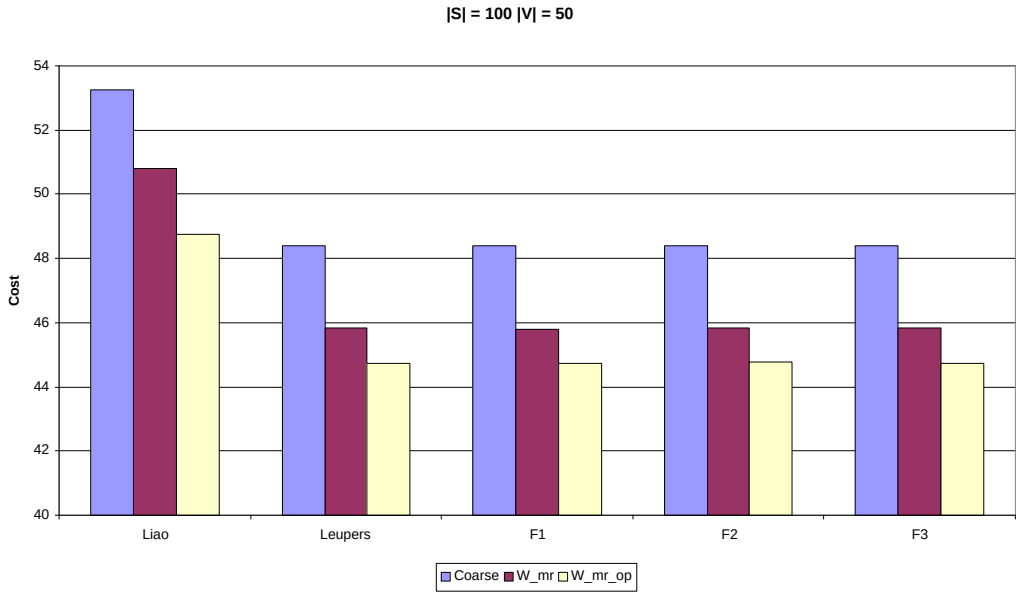


Figure 3.9: Results for SOA and SOA_mr with $|S| = 100, |V| = 50$.

Figure 3.12, 3.13, 3.14 show that our GOA_FRQ algorithm outperforms Leupers' in many cases. Especially in Figure 3.12, we can witness that beyond certain threshold, our algorithm keeps its performance stable. However, Leupers' algorithm tries to use as many ARs as possible, which makes performance of his algorithm deteriorated as the number of ARs grows. Line graphs in Figure 3.12, 3.13, 3.14 show that our mr optimization technique is beneficial, and that 2 ARs configuration always outperforms ar_mr_op as we mentioned earlier.

3.6 Chapter Summary

We have proposed a new approach of introducing a weight adjustment function and showed that its experimental results are slightly better and at least as well as the results of

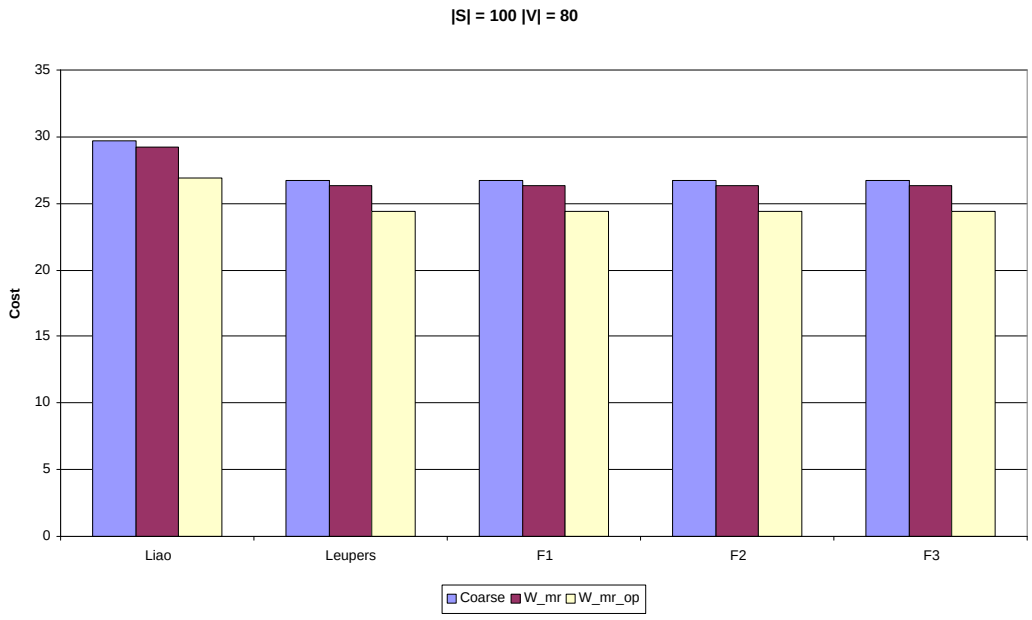


Figure 3.10: Result for SOA and SOA_mr with $|S| = 100, |V| = 80$.

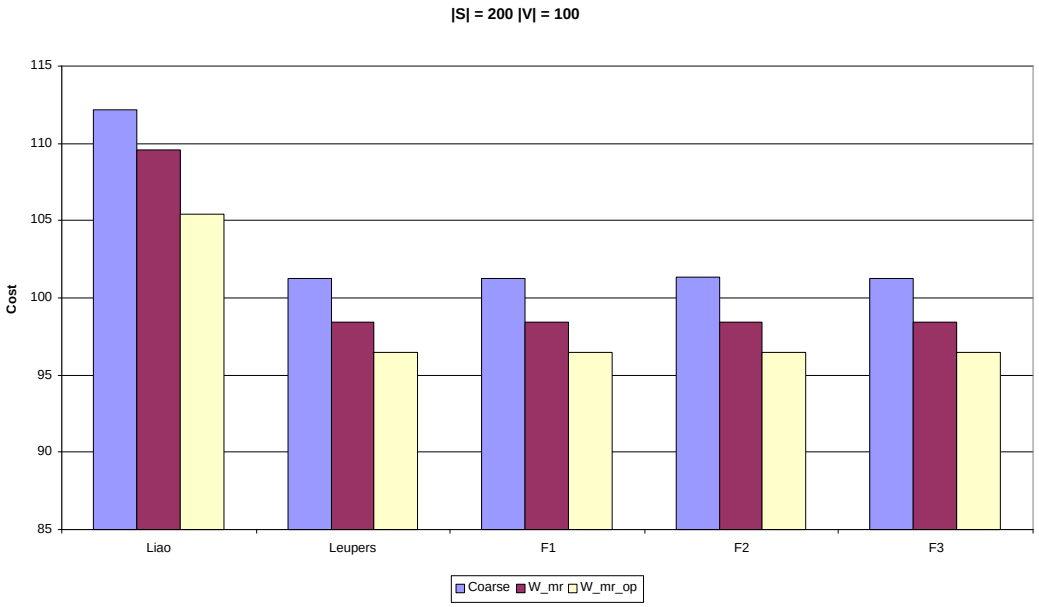


Figure 3.11: Results for SOA and SOA_mr with $|S| = 200, |V| = 100$.

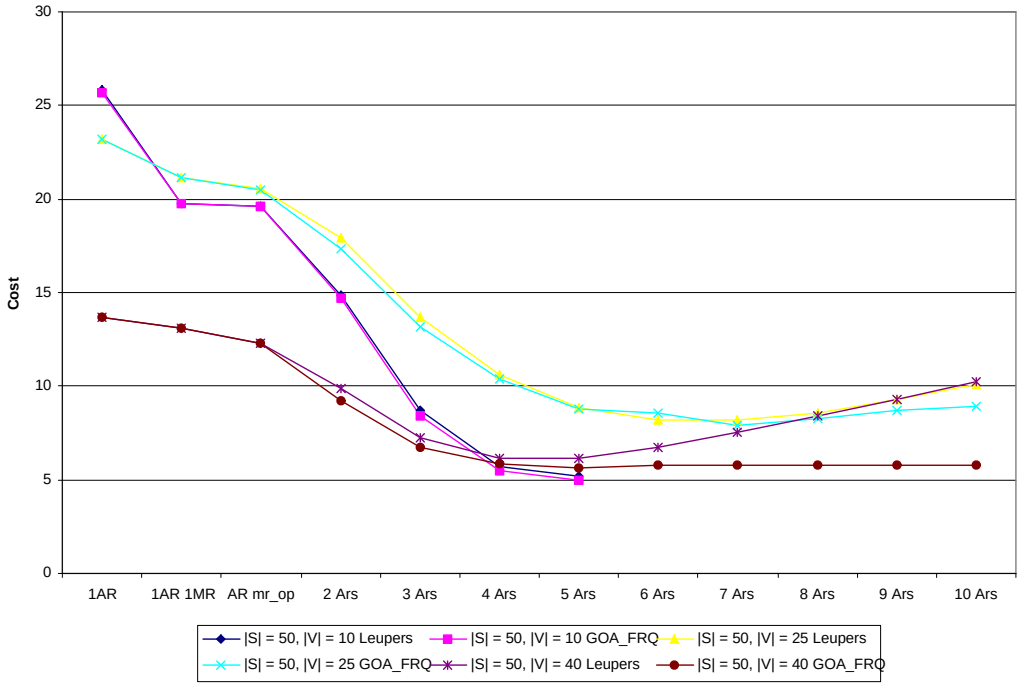


Figure 3.12: Results for GOA_FRQ.

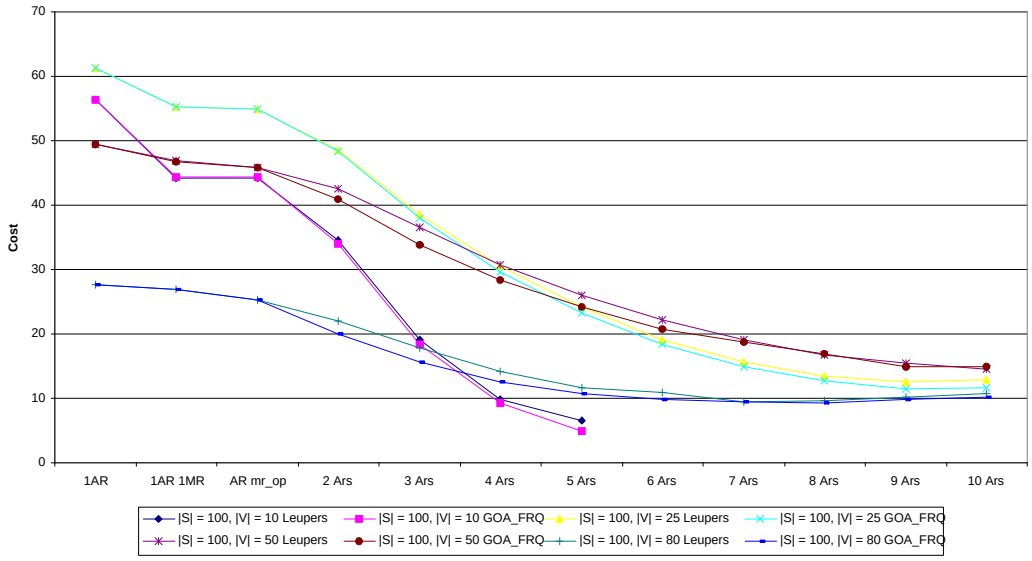


Figure 3.13: Results for GOA_FRQ.

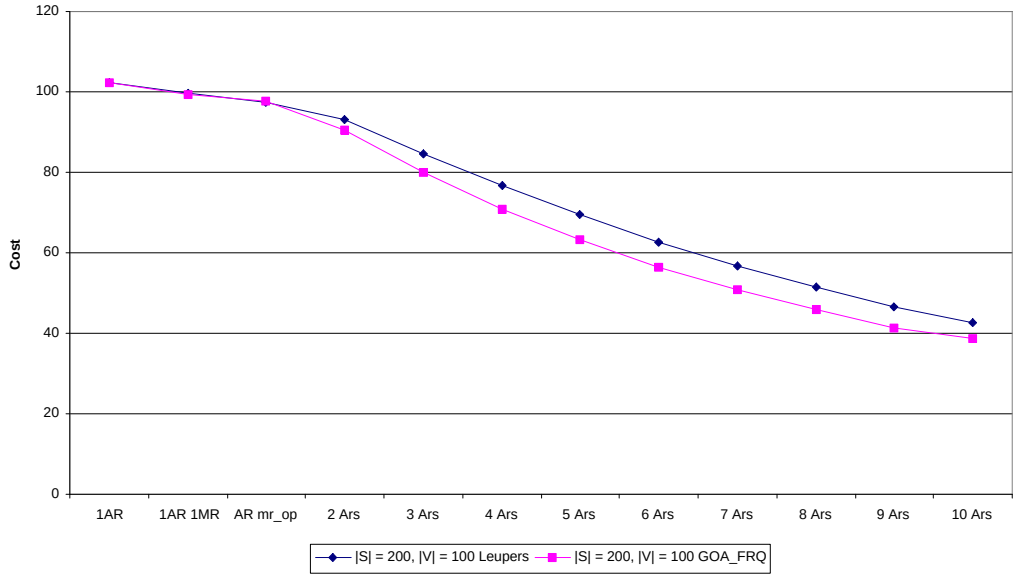


Figure 3.14: Results for GOA_FRQ.

the previous works. More importantly, we have introduced a new way of handling the same edge weight in an access graph.

As the SOA algorithm generates several fragmented paths, we show that the optimization of these path partitions is crucial to achieve an extra gain, which is clearly captured by our experimental results.

We also have proposed usage of frequencies of variables in a GOA problem. Our experimental results show that this straightforward method is better than the previous research works.

In our weight adjustment functions, we handled Preference and Interference uniformly. We applied our weight adjustment functions to random data. Real-world algorithms, however, may have some patterns that are unique to each specific algorithm. We think that we may get a better result by introducing tuning factors an then handling Preference and Interference differently according to the pattern or the regularity in a specific algorithm. For example, when $(\alpha \cdot \text{Preference})/(\beta \cdot \text{Interference})$ is used as a weight adjustment function, setting $\alpha = \beta = 1$ gives our original weight adjustment functions. Finding optimal values of tuning factors may requires exhaustive simulation and take a lot of execution time for each algorithm.

Table 3.1: The result of SOA and SOA_mr with 1000 iterations.

Size	AGU Conf.	Liao	Leupers	F_1	F_2	F_3
$\|S\| = 10$	Coarse	2.190	1.920	1.920	1.919	1.919
$\|V\| = 5$	W_mr	1.559	1.606	1.604	1.610	1.614
	W_mr_op	1.480	1.578	1.578	1.584	1.585
$\|S\| = 20$	Coarse	5.333	5.262	5.261	5.293	5.295
$\|V\| = 5$	W_mr	3.160	3.290	3.290	3.268	3.255
	W_mr_op	3.119	3.270	3.275	3.260	3.235
$\|S\| = 20$	Coarse	5.591	4.983	4.983	4.982	4.982
$\|V\| = 15$	W_mr	5.108	4.566	4.550	4.546	4.563
	W_mr_op	4.617	4.217	4.209	4.204	4.210
$\|S\| = 50$	Coarse	24.449	24.220	24.12	24.119	24.104
$\|V\| = 10$	W_mr	18.819	18.686	18.693	18.764	18.719
	W_mr_op	18.622	18.591	18.606	18.688	18.636
$\|S\| = 50$	Coarse	14.255	12.751	12.751	12.747	12.747
$\|V\| = 40$	W_mr	13.703	12.227	12.221	12.215	12.222
	W_mr_op	12.699	11.403	11.404	11.397	11.399
$\|S\| = 100$	Coarse	55.777	55.361	55.323	55.569	55.560
$\|V\| = 10$	W_mr	43.108	43.850	43.129	43.210	43.201
	W_mr_op	43.660	43.580	43.105	43.196	43.179
$\|S\| = 100$	Coarse	53.252	48.392	48.395	48.417	48.388
$\|V\| = 50$	W_mr	50.801	45.845	45.806	45.845	45.827
	W_mr_op	48.758	44.741	44.716	44.773	44.752
$\|S\| = 100$	Coarse	29.650	26.661	26.661	26.662	26.661
$\|V\| = 80$	W_mr	29.180	26.340	26.320	26.280	26.310
	W_mr_op	26.867	24.376	24.371	24.362	24.373
$\|S\| = 200$	Coarse	112.200	101.287	101.289	101.300	101.265
$\|V\| = 100$	W_mr	109.610	98.456	98.445	98.430	98.429
	W_mr_op	105.392	96.491	96.478	96.492	96.477

Table 3.2: The result of GOA with 500 iterations.

AGU Conf.	$\|S\| = 50, \|V\| = 10$		$\|S\| = 50, \|V\| = 25$	
	Leupers	GOA_FRQ	Leupers	GOA_FRQ
1AR	25.840	25.680	23.232	23.232
1AR_1MR	19.756	19.760	21.134	21.120
ARmr_op	19.638	19.634	20.526	20.506
2 ARs	14.856	14.722	17.942	17.338
3 ARs	8.708	8.410	13.714	13.158
4 ARs	5.714	5.466	10.642	10.420
5 ARs	5.220	4.978	8.890	8.806
6 ARs			8.200	8.540
7 ARs			8.200	7.916
8 ARs			8.590	8.246
9 ARs			9.278	8.712
10 ARs			10.106	8.908
AGU Conf.	$\|S\| = 50, \|V\| = 40$		$\|S\| = 100, \|V\| = 10$	
	Leupers	GOA_FRQ	Leupers	GOA_FRQ
1AR	13.710	13.710	56.356	56.326
1AR_1MR	13.132	13.128	44.210	44.318
ARmr_op	12.294	12.292	44.196	44.300
2 ARs	9.910	9.228	34.498	33.984
3 ARs	7.254	6.742	19.160	18.312
4 ARs	6.180	5.862	9.808	9.328
5 ARs	6.126	5.606	6.460	5.000
6 ARs	6.768	5.814		
7 ARs	7.542	5.814		
8 ARs	8.402	5.814		
9 ARs	9.326	5.814		
10 ARs	10.266	5.814		

Table 3.3: The result of GOA with 500 iterations (continued.)

AGU Conf.	$\|S\| = 100, \|V\| = 25$		$\|S\| = 100, \|V\| = 50$	
	Leupers	GOA_FRQ	Leupers	GOA_FRQ
1AR	61.252	61.240	49.442	49.448
1AR_1MR	55.324	55.300	46.828	46.802
ARmr_op	54.954	54.944	45.758	45.764
2 ARs	48.618	48.326	42.508	40.892
3 ARs	38.612	37.918	36.560	33.894
4 ARs	30.478	29.674	30.672	28.332
5 ARs	24.190	23.282	25.982	24.112
6 ARs	19.120	18.282	22.178	20.694
7 ARs	15.648	14.908	19.126	18.740
8 ARs	13.512	12.722	16.796	16.940
9 ARs	12.480	11.476	15.460	14.916
10 ARs	12.840	11.600	14.504	14.940
AGU Conf.	$\|S\| = 100, \|V\| = 80$		$\|S\| = 200, \|V\| = 100$	
	Leuper	GOA_FRQ	Leuper	GOA_FRQ
1AR	27.642	27.642	102.444	102.444
1AR_1MR	26.996	26.994	99.514	99.494
ARmr_op	25.260	25.250	97.540	97.542
2 ARs	21.988	19.996	93.260	90.576
3 ARs	17.768	15.568	84.482	79.906
4 ARs	14.156	12.634	76.722	70.846
5 ARs	11.602	10.722	69.458	63.264
6 ARs	10.840	9.766	62.752	56.430
7 ARs	9.514	9.446	56.736	50.696
8 ARs	9.666	9.344	51.560	45.774
9 ARs	10.102	9.732	46.600	41.414
10 ARs	10.814	10.190	42.542	38.820

CHAPTER 4

ADDRESS REGISTER ALLOCATION IN DSPS

Most signal processing algorithms have a small number of core processing tasks that are to be implemented by loop statements in which several simple operations are applied to a massive amount of signals (data). The loops have large number of iterations. So, it is very crucial in signal processing to optimize the code inside the loops. Usually massive data are stored in arrays, which are considered as a convenient data structure especially in a loop. In most programs addressing computation accounts for a large fraction of the execution time. In general purpose programs, over 50% of the execution time is for addressing, and 1 out of 6 instructions is an address manipulation instruction [37]. From the fact that typical DSP programs access massive amounts of data, it is easy to conclude that handling addressing computation properly in DSP domain is a more important subject than in general purpose computing in order to achieve a compact code with real-time performance. The DSP processors have limited number of addressing modes. References to arrays should be translated into indirect addressing mode using ARs. In order to reduce the number of explicit address register instructions, array references should be carefully assigned to address registers.

4.1 Related Work on Address Register Allocation

The first algorithm for optimal allocation of index register for addressing operation was proposed by [39]. Several research works have been done on addressing modes for DSP architectures [37, 58, 4, 5, 6, 56, 13]. Araujo [4, 5, 6] insists that efficient usage of the AGU needs two tasks, identification of an addressing mode and allocation of address registers to addressing operations. First, he allocates virtual address registers to pointer variables and array references, and then allocates physical registers to the virtual address registers. He defines an Array Indexing Allocation Problem as a problem of allocating virtual AR to array references and proposes a solution by introducing an Indexing Graph (IG). Vertices in IG are array references and edges represent the possible transition from one array access to another array access without an address instruction. The goal of IG is to allocate the minimum number of ARs by maximizing the number of array accesses that can share an AR. He formulates an IG covering problem as finding the disjoint path/cycle cover of IG which minimize the total number of paths and cycles. IG covering is NP-hard. So, he simplifies IG covering by dropping cycles from it. Actually it is a minimum vertex-disjoint path covering (MDPC) problem of a graph. He solves his simple IG covering by using Hopcroft-Karp's solution [38] of the bipartite matching problem. His simple IG covering can not eliminate need for explicit address instructions in the loop body by ignoring cycles. In embedded processing, it is not unusual that some simple operations are applied to a huge amount of data in regularly and massively repeated manner. Eventually, the accumulated effects of explicit address instructions in a loop can not be and should not be ignored.

Leupers et al. [56, 13] defines an AR allocation problem as finding a minimum path cover of a distance graph $G = (V, E)$ such that all nodes in G are touched by exactly one

path and for each path, the distance between a head and a tail of the path is within a maximum modify range. Leupers introduces an extended distance graph $G' = (V', E'), V' = V \cup \{a'_1, \cdots, a'_n\}$ in which each node $a'_i \notin V$ represents the array reference a_i in the next loop iteration. His extended graph captures the possibilities of address instruction free transitions from one array reference in the current iteration to the array reference in the next iteration. He assigns a unit weight to each edge in an extended distance graph and then tries to find the longest path from a_i to a'_i. He uses Araujo's matching-based algorithm [6] of simple IG covering to find a lower bound L on the number of ARs, and his own path-based algorithm to find an upper bound U. He puts these two algorithms into his branch-and-bound algorithm to find an optimum solution to the AR allocation problem. After finding a lower bound and an upper bound, he selects feasible edge $e = (a_i, a_j)$. An edge $e = (a_i, a_j)$ is feasible if and only if there is a path $(a_j, \cdots, a'_i)$ in the extended graph. He constructs two distance graphs, $G_{\bar{e}}$ and G_e. $G_{\bar{e}}$ excludes the feasible edge e and G_e includes the edge e by merging two nodes a_i and a_j into one node. He computes lower bounds $L_{\bar{e}}$ for $G_{\bar{e}}$ and L_e for G_e by using matching-based algorithm. If $L_{\bar{e}} > U$, all solutions for $G_{\bar{e}}$ can not be optimal and edge e must be included. If $L_e > U$, all solutions for G_e can not be optimal and e must be excluded. He recursively applies his branch-and-bound algorithm to find minimum number of ARs. His algorithm can find optimal solution to AR allocation problem. However, his recursive branch-and-bound algorithm contains two different algorithms to find a lower bound and an upper bound, and for each feasible edge e two different distance graph are constructed and tested recursively. His algorithm has an exponential time complexity and is unnecessarily complicated.

4.2 Address Register Allocation

Given an array reference sequence, the address register allocation problem is one of partitioning the array references into groups in such a way that array references in any group are to be assigned to the same address register with the objective of minimizing the total number of explicit address instructions by taking advantage of the AGU's auto-increment/decrement capability.

Figure 4.1-(a) shows two statements in a loop. Figure 4.1-(b) shows a corresponding array reference sequence. For simplicity, only the address register instructions are shown in the figure. In Figure 4.1-(c), only one address register, AR0 is used for all five array references of an array, **A**. Except the initialization instruction, three explicit AR instructions are needed in each iteration of the loop. When the loop repeats many times (a loop bound N is large), it deteriorate not only the size of the code but also the execution speed. In Figure 4.1-(d), and (e), two ARs, AR0 and AR1, are used. In Figure 4.1-(d), the first three references are assigned to AR0, and the last two to AR1. Except two initialization instructions, three explicit address register instructions are still needed, even though two ARs are employed. On the contrary, in Figure 4.1-(e), the first, third, and fifth references are assigned to AR0, and the second and fourth to AR1. There are no explicit AR instructions in the loop, which is a huge gain for the speed when N is large, and also high-quality compact code is generated. We propose an algorithm to eliminate explicit AR instructions in a loop statement, and also propose a quick algorithm to find the lower bound on the number of ARs. Figure 4.1 shows that while carefully chosen array register allocation can eliminate explicit address instructions, assigning wrong array references to ARs may requires explicit address instructions despite of using multiple ARs.

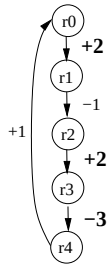
```
for(i=1;i<=N;i++) {
  A[i+1] = A[i] + A[i+2]
  A[i] = A[i+3]
}
```

(a) code

```
for(i=1;i=<N;i++) {
  A[i]      //ref_0
  A[i+2]    //ref_1
  A[i+1]    //ref_2
  A[i+3]    //ref_3
  A[i]      //ref_4
}
```

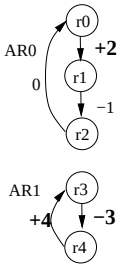
(b) an array access sequence

```
LDAR AR0, &A[1]
for(i=1;i<=N;i++) {
  ADAR AR0, 2
  *(AR0)−−
  ADAR AR0,2
  SBAR AR0,3
  *(AR0)++
}
```



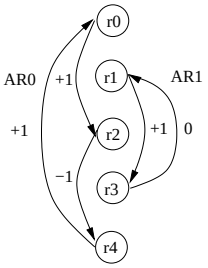
(c) one AR used

```
LDAR AR0, &A[1]
LDAR AR1, &A[4]
for(i=1;i<=N;i++) {
  ADAR AR0, 2
  *(AR0)−−
  SBAR AR1, 3
  ADAR AR1, 4
}
```



(d) unoptimized AR0:r0,r1,r2 AR1:r3,r4

```
LDAR AR0, &A[1]
LDAR AR1, &A[3]
for(i=1;i<=N;i++) {
  *(AR0)++
  *(AR1)++
  *(AR0)−−
  *(AR1)
  *(AR0)++
}
```



(e) optimized AR0:r0,r2,r4 AR1:r1,r3

Figure 4.1: An example of AR allocation.

4.3 Our Algorithm

Figure 4.2 shows an array reference sequence of a program where a_0 is the first array reference and a_{r-1} is the last one, and l is an index control variable. We assume that each array reference $a_i,\ 0 \leq i < r$ is of the form $l \pm c_i$, where c_i is a constant.

for $l = L$ **to** U **do**
  a_0
  a_1
  $\vdots$
  a_{r-1}
enddo

Figure 4.2: Basic structure of a program.

Definition 4.1 *A function* $offset(a_i)$ *returns an offset* $\pm c_i$ *of an array reference* $a_i = l \pm c_i$.

Definition 4.2 *A distance graph is* $G^M(V, E)$, *where* $V = \{a_i | 0 \leq i < r\}$, *and* $E = \{(a_i, a_j) | 0 \leq i < j < r, |offset(a_i) - offset(a_j)| \leq M\}$. *An edge* $e = (a_i, a_j)$ *is called a forward edge because a source* a_i *precedes a destination* a_j *in an array reference sequence.* M *is a maximum modify range. A distance graph can be called as a forward edge graph, either*

When the difference of offsets of two different array references $a_i, a_j, i < j$ is less than or equals to M, the transition from a_i to a_j can be done without explicit address instructions. A distance graph is an acyclic directed graph because direction of all edges is from a node a_i to a node a_j; A node a_i precedes a node a_j in an array reference sequence. Figure 4.3 shows a distance graph for the array reference sequence in Figure 4.1.

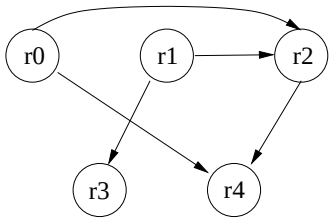


Figure 4.3: A distance graph.

Definition 4.3 *A back edge graph* $G_B(V_B, E_B)$ *consists of* $V_B = \{a_i | 0 \leq i < r\}, E_B = \{(a_j, a_i) | 0 \leq i < j < r, |offset(a_i) - offset(a_j) +$ *iteration step*$| \leq M\}$. *An edge* $e = (a_j, a_i)$ *is called a back edge because a destination* a_i *precedes a source* a_j *in an array reference sequence.*

When the sum of a loop trip step and the difference of offsets of two different array references $a_i, a_j, i < j$ is less than or equal to M, it is possible to update an AR that points to a reference a_j at the current iteration $k, L \leq k < U$ to a reference a_i of the $(k+1)$th iteration. In a similar manner, a back edges graph is also acyclic. Figure 4.4 shows a back edge graph that corresponds to Figure 4.1.

Definition 4.4 *An extended graph* $G'(V', E')$ *consists of* $V' = V$ *and* $E' = E \cup E_B$.

Figure 4.6-(a) shows an extended graph.

Definition 4.5 *When all the references on a path* P *can be assigned to a AR and then be referenced in the appearance order on a path* P *by the same AR without using explicit address instructions, it is said that a path* P *is covered by the AR, or a path* P *is coverable by the AR. Equivalently, all the references on the covered path or coverable path are called covered by or coverable by the AR.*

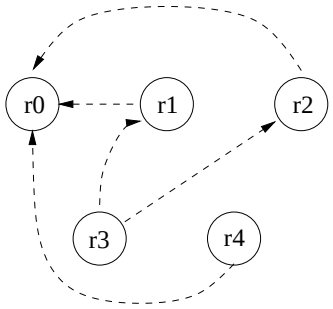


Figure 4.4: A back edge graph.

Definition 4.6 *A compatible graph* $G_c(V_c, E_c)$ *consists of* $V_c = \{P | \text{a path } P \in G^M\}, E_c = \{(P_1, P_2) | P_1 \cap P_2 = \phi\}$.

A compatible graph is a weighted undirected graph. An edge $e = (P_1, P_2)$ has a weight $|P_1| + |P_2|$, where $|P_1|$ and $|P_2|$ are the length of paths P_1 and P_2, respectively. Figure 4.5-(c) shows a compatible graph.

Lemma 4.1 *When a cycle* C *in* G' *contains exactly one back edge* $(c_{\alpha(p)}, c_{\alpha(0)})$ *and is of the form,* $C = c_{\alpha(0)}c_{\alpha(1)} \cdots c_{\alpha(p)}c_{\alpha(0)}$, $\alpha(i) < \alpha(j)$, $0 \leq \alpha(i), \alpha(j) < r, 0 \leq i < j \leq p < r$, *all the array references in a cycle* C *can be covered by the same AR.*

(Proof) In an extended graph G', the source and the destination of a forward edge can be covered by the same AR by its definition. If a cycle C is of the form $c_{\alpha(0)}c_{\alpha(1)} \cdots c_{\alpha(p)}c_{\alpha(0)}$, then a constituent path from $c_{\alpha(0)}$ to $c_{\alpha(p)}$ is coverable because the constituent path consists of only forward edges. The cycle C has only one back edge $(c_{\alpha(p)}, c_{\alpha(0)})$, which is coverable by the definition of a back edge. Therefore, all the references on the cycle C are coverable.

Lemma 4.2 *The number of Strongly Connected Components (SCC) of an extended graph* G' *is a lower bound of the number of address registers of AR allocation problem.*

(Proof) Let a_i and a_j be two different array references. Assume that a_i and a_j belong to different SCCs. If there is a coverable cycle in G' that contains both of a_i and a_j, then a_i and a_j will belong to the same SCC by the definition of SCC. It is a contradiction of the assumption. So, there is no coverable cycle that contains a_i and a_j. A SCC may contain more than one back edges. In that case, it can not be guaranteed that the array references in the SCC are covered by one AR. A SCC requires at least one AR in order for the array references in the SCC to be covered. Therefore, the number of SCC in G' is a lower bound of the number of address registers.

We propose an algorithm to eliminate explicit AR instructions in a loop, and also propose a quick algorithm to compute the lower bound on the number of ARs by finding SCCs in an extended graph. Figure 4.5 shows our proposed algorithm.

Figure 4.6-(a) shows an extended graph, in which solid lines represent forward edges and dotted lines represent back edges, that corresponds to a problem in Figure 4.1. The idea behind our algorithm is that after constructing an extended graph, all paths from v_a to v_b for each back edge, (v_b, v_a) are found, and then a compatible graph is constructed from the paths, in which nodes are paths, and if two paths are disjoint, then there is an edge between those two nodes whose weight is the sum of lengths of each path. Figure 4.6-(b) shows paths for each back edge. Figure 4.6-(c) is a compatible graph. The largest weighted edge is selected. In Figure 4.6-(c), the edge between two paths, $(0, 2, 4)$ and $(1, 3)$, has the largest weight. The first, third, and fifth references are assigned to one AR, and the second, and fourth to another AR. Each selected edge requires two address registers. The larger the weight of the selected edge is, the more array references are covered by two ARs. Until all array references are assigned, the procedure of selecting the largest one and then updating a corresponding extended graph is repeated.

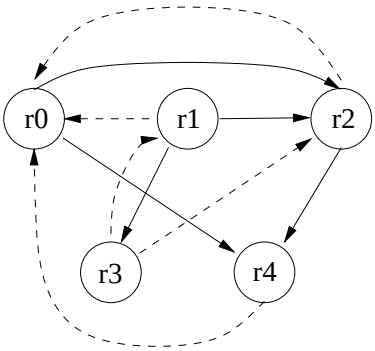
Procedure AR Allocation(Seq)
Seq : an array reference sequence
{
Make a distance graph from Seq
Find all back edges

$i \leftarrow 0$
while ($|back\_edges| > 0$) **do**
  Find all the paths from v_a to v_b for each back edge,$e = (v_b, v_a)$
  Construct compatible graph
  $AR[i] \leftarrow$ choose the larger one between the largest compatible edge and longest path
  $i \leftarrow i + 1$
  $Seq \leftarrow Seq - \{v | v \in AR\}$
  Update a distance graph and back edges
enddo

while ($|Seq| > 0$) **do**
  $v \leftarrow$ a reference from Seq
  $Seq \leftarrow Seq - \{v\}$
  $AR[i] \leftarrow v$
  $i \leftarrow i + 1$
enddo

return AR;
}

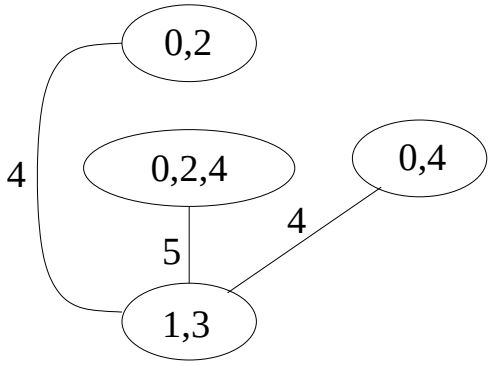
Figure 4.5: Our AR Allocation Algorithm.



(a) an extended graph

<2,0> 0 −> 2
<1,0> none
<4,0> 0 −> 2 −> 4, 0 −> 4
<3,1> 1 −> 3
<3,2> none

(b) all paths for each back edge



(c) a compatible graph

Figure 4.6: An example of our algorithm.

Table 4.1: The result of AR allocation with 100 iterations for $|D| = 1$ and $|D| = 2$.

		$\|D\| = 2$				$\|D\| = 3$			
		$M = 1$		$M = 2$		$M = 1$		$M = 2$	
$n = 5$	AR	2.13	42.60	1.52	30.40	2.52	50.40	1.80	36.00
	SCC	1.96	39.20	1.20	24.00	2.39	47.80	1.45	29.00
	$\frac{\text{AR}}{\text{SCC}}$(%)	108.67		126.67		105.44		124.14	
$n = 8$	AR	2.22	27.75	1.71	21.38	3.03	37.88	1.99	24.88
	SCC	1.81	22.62	1.01	12.62	2.64	33.00	1.18	14.75
	$\frac{\text{AR}}{\text{SCC}}$(%)	122.65		169.31		114.77		168.64	
$n = 10$	AR	2.29	22.90	1.68	16.80	3.27	32.70	2.12	21.20
	SCC	1.60	16.00	1.00	10.00	2.54	25.40	1.16	11.60
	$\frac{\text{AR}}{\text{SCC}}$(%)	143.12		168.00		128.74		182.76	
$n = 12$	AR	2.55	21.25	2.12	17.67	3.34	27.83	2.53	21.08
	SCC	1.41	11.75	1.00	8.33	2.22	18.50	1.08	9.00
	$\frac{\text{AR}}{\text{SCC}}$(%)	180.85		212.00		150.45		234.26	
$n = 15$	AR	2.73	18.20	2.08	13.87	3.60	24.00	2.85	19.00
	SCC	1.16	7.73	1.00	6.67	2.04	13.60	1.03	6.87
	$\frac{\text{AR}}{\text{SCC}}$(%)	235.34		208.00		176.47		276.70	
$n = 17$	AR	2.93	17.24	2.29	13.47	3.60	21.18	2.98	17.53
	SCC	1.11	6.53	1.00	5.88	1.67	9.82	1.03	6.06
	$\frac{\text{AR}}{\text{SCC}}$(%)	263.96		229.00		215.57		289.32	
$n = 20$	AR	3.26	16.30	2.62	13.10	3.83	19.15	3.13	15.65
	SCC	1.07	5.35	1.00	5.00	1.43	7.15	1.02	5.10
	$\frac{\text{AR}}{\text{SCC}}$(%)	304.67		262.00		267.83		306.86	

Table 4.2: The result of AR allocation with 100 iterations for $|D| = 3$ and $|D| = 4$.

		$\|D\| = 4$				$\|D\| = 5$			
		$M = 1$		$M = 2$		$M = 1$		$M = 2$	
$n = 5$	AR	3.00	60.00	2.31	46.20	3.41	68.20	2.59	51.80
	SCC	2.99	59.80	1.93	38.60	3.39	67.80	2.32	46.40
	$\frac{\text{AR}}{\text{SCC}}$(%)	100.33		119.69		100.59		111.64	
$n = 8$	AR	3.79	47.38	2.56	32.00	4.29	53.62	2.94	36.75
	SCC	3.56	44.50	1.66	20.75	4.15	51.88	2.12	26.50
	$\frac{\text{AR}}{\text{SCC}}$(%)	106.46		154.22		103.37		138.68	
$n = 10$	AR	3.92	39.20	2.67	26.70	4.51	45.10	3.11	31.10
	SCC	3.42	34.20	1.59	15.90	4.16	41.60	1.82	18.20
	$\frac{\text{AR}}{\text{SCC}}$(%)	114.62		167.92		108.41		170.88	
$n = 12$	AR	4.04	33.67	2.88	24.00	4.75	39.58	3.36	28.00
	SCC	3.21	26.75	1.29	10.75	4.09	34.08	1.64	13.67
	$\frac{\text{AR}}{\text{SCC}}$(%)	125.86		223.26		116.14		204.88	
$n = 15$	AR	4.23	28.20	3.19	21.27	5.19	34.60	3.49	23.27
	SCC	2.87	19.13	1.20	8.00	3.80	25.33	1.48	9.87
	$\frac{\text{AR}}{\text{SCC}}$(%)	147.39		265.83		136.58		235.81	
$n = 17$	AR	4.28	25.18	3.32	19.53	5.15	30.29	3.65	21.47
	SCC	2.59	15.24	1.10	6.47	3.80	22.35	1.37	8.06
	$\frac{\text{AR}}{\text{SCC}}$(%)	165.25		301.82		135.53		266.42	
$n = 20$	AR	4.56	22.80	3.48	17.40	5.49	27.45	3.76	18.80
	SCC	2.24	11.20	1.06	5.30	3.33	16.65	1.25	6.25
	$\frac{\text{AR}}{\text{SCC}}$(%)	203.57		328.30		164.86		300.80	

4.4 Experimental Results

We experiment our heuristics with different scenarios. We repeat each experiment 100 times. Tables 4.1 and 4.2 show the experimental results. The first column shows the length of an array reference sequence. The second column is the results of $|D| = 2$. D is a maximum offset difference. When $|D| = 2$, the array reference offset, c_i is between -2 and 2. The first and second sub-columns of the second column are the results of $M = 1$ and $M = 2$. M is a maximum modify range. Each row shows the results of the number of ARs, of a lower bound, and the percentage ratio of the number of ARs to the number of SCCs. When $n = 5, |D| = 2$,and $M = 1$, the results show that 2.13 ARs are needed and a lower bound is 1.96. The percentage ratio of ARs to SCCs is 108.67%. This ratio shows that the number of ARs is very close to a lower bound. The percentage ratio of the number of ARs to the length of an array reference sequence is 42.6%, and the percentage ratio of the number of SCCs to the length of array reference sequence is 39.2%. When a maximum modify range is 2, the extended graph becomes more dense than when a modify range is 1. The numbers of ARs and SCCs are 1.52 and 1.2 respectively, which are better results. The percentage ratio of ARs to SCCs is 126.67% , which is worse than 108.67%.

A larger modify range introduces more forward edges and also more back edges. More forward edges contribute to the better result of ARs, and more back edges contribute to the better result of SCCs. When an array reference sequence becomes longer, more ARs are needed. When $n = 20, |D| = 2$, and $M = 1$, 3.26 ARs are needed. However, the percentage ratio of AR to the length of an array reference sequence drops from 42.6% to 16.3%. As an array reference sequence becomes longer, the number of potential forward edges grows geometrically because when the length of an array reference sequence is n, the extended graph may have $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ forward edges maximally. Also the number

of potential back edges is as same. When n becomes larger, our lower bound of SCCs tends to be too optimistic. For example, when $n = 20, |D| = 2$, and $M = 1$, there are only 1.07 SCCs in an extended graph. We think it is because newly introduced back edges constitute a larger cycles, which deteriorates the closeness of our lower bound.

We repeat our experiment with several maximum offset differences, $|D| = 3, 4, 5$. In each case, the same trends we mentioned so far are observed. When $|D|$ becomes larger, the experimental results become worse as expected. For example, when $n = 5, |D| = 3$, and $M = 1$, 2.52 ARs are needed, and a lower bound is 2.39. Both of them are worse results than when $|D| = 2$.

4.5 Chapter Summary

We have developed an algorithm that can eliminate the explicit use of address register instructions in a loop. By introducing a compatible graph, our algorithm tries to find the most beneficial partitions at the moment. In addition, we developed an algorithm to find a lower bound on the number of ARs by finding the strong connected components (SCCs) of an extended graph.

We implicitly assume that unlimited number of ARs are available in the AGU. However, usually it is not the case in real embedded systems in which only limited number of ARs are available. Our algorithm tries to find partitions of array references in such a way that ARs cover as many array references as possible, which leads to minimization of the number of ARs needed. With the limited number of ARs, when the number of ARs needed to eliminate the explicit use of AR instructions is larger than the number of ARs available in the AGU, it is not possible to eliminate AR instructions in a loop. In that case, some partitions of array references should be merged in a way that the merger should minimize the number of explicit use of AR instructions. Our future works will be finding a model that

can capture the effects of merging partitions on the explicit use of AR instructions. Based on that model, we will find efficient solution of AR allocation with the limited number of ARs.

When an array reference sequence becomes longer, and then the corresponding extended graph becomes denser, our lower bound on ARs with SCCs tended to be too optimistic. To prevent the lower bound from being too optimistic, we need to drop some back edges from the extended graph. In that case, it will be an important issue to determine which back edges should be dropped, which will be a focus of our future work.

CHAPTER 5

REDUCING MEMORY REQUIREMENTS VIA STORAGE REUSE

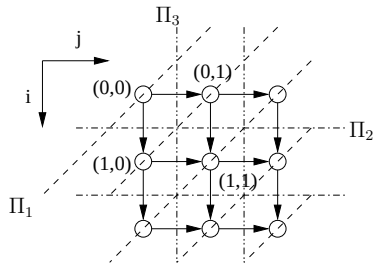
Each algorithm has its own data dependence relations. Data dependences impose fundamental ordering constraints on a program that implements the algorithm. Our target application domain - embedded processing - has some features that distinguish it from general application domain. Some simple operations will be applied to massive amount of data in repeated manner. Those computational patterns are usually time-invariant (static). Those kinds of static computation patterns can be easily implemented in a loop. Especially regarding with huge amount of repeated computations on a massive amount of data in a special purpose processing domain, a loop is very useful program structure. Iteration Space Dependence Graph (ISDG) [82] is a useful representation to capture dependences. A vertex in an ISDG represents a computation in an iteration and an edge represents a dependence from a source iteration to a destination iteration. A k nested loop is represented by k-dimensional ISDG. An instance of a computation in a loop is represented by k-dimensional vector, in which ith vector element corresponds to ith innermost index value in a loop.

Anti-dependence and output dependence can be eliminated by scalar-renaming and array expansion [25, 9], but it requires extra memory for the expense. A scheduling determines which computation will be executed in which time step. A scheduling is to make

ordering of computations, which imposes some computations to precede other computations. A schedule should not violate dependence relations. The integrity of an algorithm is to be maintained by obeying its computational ordering constraints (dependence relations). A legal schedule should satisfy dependence relations.

5.1 Interplay between Schedules and Memory Requirements

In this chapter, we assume that dependence relations are regular and static, and loop transformations were already applied. So, we do not apply loop transformation techniques. The legality condition of a schedule is defined by expressing its respect for dependence relations of a given problem. Dependence relations impose a legality constraint on a schedule. A schedule affects the amount of memory requirements of computations in a loop. We may infer that dependence relations are closely linked with memory requirements through a schedule. Figure 5.1 shows a simple ISDG, in which there are two dependence, $\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. When we use a schedule $\Pi_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ in this example, all the computations in j axis will be executed in the same time step. However, in this case a computation in $(0, 1)$ iteration depends on the result produced by a computation in $(0, 0)$. So, scheduling $(0, 1)$ and $(0, 0)$ into the same time step violates this dependence. With the very same reason, a schedule $\Pi_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ is not valid, either. $\Pi_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ obeys both of dependences. Π_1 is a legal schedule. There might be more than one legal schedules. In that case, it is a very important issue to find the best schedule. We will formally define the legality condition of a schedule and its optimality from the perspectives of the memory requirements, of completion time and also from the perspective of combination of memory requirements and completion time.

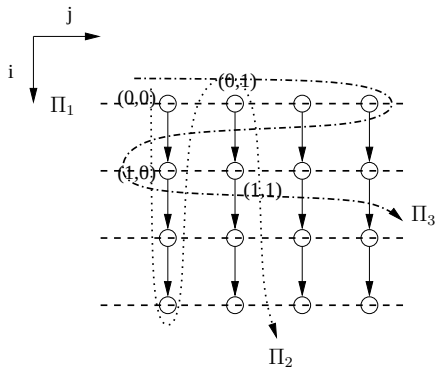


$\Pi_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ $\Pi_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $\Pi_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

Figure 5.1: A simple ISDG example.

Figure 5.2 shows inter-relationships between memory requirements and a schedule. There is one dependence $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Let $|N_i|$ and $|N_j|$ be the size of i-axis and j-axis respectively. Under a schedule $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $|N_j|$ memory locations are needed. It takes $O(|N_i|)$ time to complete computations. With a schedule $\begin{pmatrix} 1 \\ |N_i| \end{pmatrix}$, one memory location is needed, and $O(|N_i||N_j|)$ time is required. A schedule $\begin{pmatrix} |N_j| \\ 1 \end{pmatrix}$ requires $|N_j|$ memory locations and $O(|N_i||N_j|)$ time.

There are some interesting observations on the relations among a dependence, a schedule, and memory requirements in this example. To make the observation clear, let us assume that $|N_i|$ and $|N_j|$ be same or their difference be a constant ($|N_i| \approx |N_j|$). There are $|N_i||N_j|$ computations in this ISDG. The schedules $\begin{pmatrix} 1 \\ |N_i| \end{pmatrix}$ and $\begin{pmatrix} |N_j| \\ 1 \end{pmatrix}$ are sequential. So, their completion time is same as $O(|N_i||N_j|)$. The interesting thing is that their memory requirements are dramatically different. The difference comes from a dependence vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. A sequential schedule Π_2 makes ordering of computations along



Schedule	Memory	Time
$\Pi_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\|N_j\|$	$\|N_i\|$
$\Pi_2 = \begin{pmatrix} 1 \\ N_i \end{pmatrix}$	$\|1\|$	$\|N_i\|\|N_j\|$
$\Pi_3 = \begin{pmatrix} N_j \\ 1 \end{pmatrix}$	$\|N_j\|$	$\|N_i\|\|N_j\|$

Figure 5.2: Memory requirements and completion time with different schedules.

a dependence vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. However, another sequential schedule Π_3 does not follow a dependence vector.

Definition 5.1 *When a memory location used in one iteration c_1 is reusable by another iteration c_2 without affecting other computations that depend on the value in an iteration c_1, a difference vector $(c_2 - c_1)$ is called a storage vector or an occupancy vector.*

Definition 5.2 *When all the iterations along a storage vector can share a same memory location under a schedule, it is said that the storage vector is respected by the schedule or that the schedule respects the storage vector.*

In Figure 5.2, a dependence vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is a storage vector because computations along the dependence vector can share memory location. We already know that a schedule affects memory requirements. Now, in Figure 5.2, we observe that the inter-relation between a schedule and a storage vector also affects the amount of memory requirements. As we can see in Figure 5.2, whether or not a storage vector is taken as advantage to share memory depends on inter-relations between a schedule and a storage vector. Obviously, when a schedule takes advantage of a storage vector, computations along the storage vector share memory locations, which will lead to reduce memory requirements.

Definition 5.3 *When a storage vector is respected by any legal schedules, it is called a universal storage vector or a universal occupancy vector (UOV).*

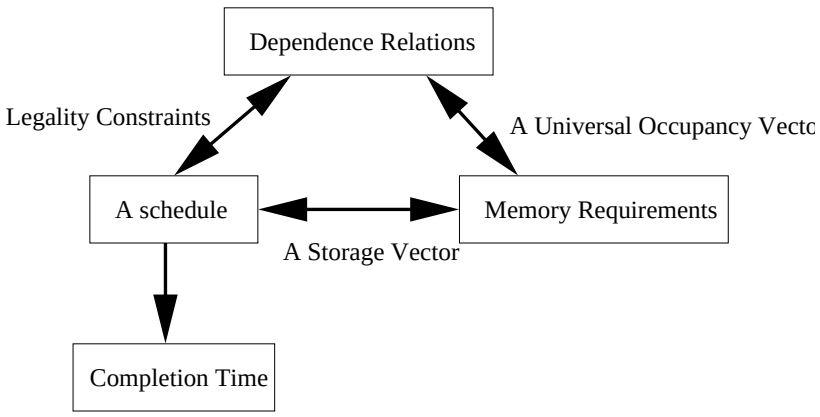


Figure 5.3: Inter-relations.

Figure 5.3 summarizes the inter-relations among dependence relations, a schedule, memory requirements, and completion time. The arrows in Figure 5.3 describe the inter-relations among corresponding factors. For example, legality constraints between dependence relations and a schedule explain that dependence relations enforce legal conditions on a schedule, and that a legal schedule should satisfy the legal condition of dependence relations. A schedule affects the amount of memory requirements. The effects of a schedule on the memory can be described by the inter-relations between a schedule and a storage vector. By the definition of a UOV, a UOV describes the direct inter-relations between dependences and memory requirements. For a UOV, any specific legal schedule is a don't-care condition. From the dependence vectors, a UOV would be found directly. A UOV sets an upper bound on the memory requirements. From this direct inter-relations, we can infer that applying some loop transformation techniques and then changing dependences may have an impact on memory requirements. However, in this chapter, we will not consider loop transformations. Strout [74] shows that determining if a vector is a UOV is a NP-complete problem. We need to define optimality of a schedule from the perspective of inter-relations among those factors as shown in Figure 5.3.

Definition 5.4 *A schedule that has the shortest completion time for a given problem is called a time-optimal schedule. When a schedule requires minimum amount of memory for a given problem, it is called a space-optimal schedule or memory-optimal schedule.*

As we can see in Figure 5.2, $\Pi_2 = \begin{pmatrix} 1 \\ N_i \end{pmatrix}$ and $\Pi_3 = \begin{pmatrix} N_j \\ 1 \end{pmatrix}$ are not time-optimal because $\Pi_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ has a shorter completion time $O(|N_i|)$. However, $\Pi_2 = \begin{pmatrix} 1 \\ N_i \end{pmatrix}$ is memory-optimal because it requires only one memory location. $\Pi_3 = \begin{pmatrix} N_j \\ 1 \end{pmatrix}$ is not memory-optimal. The problem of schedules Π_2 and Π_3 is their completion time, which is

$O(|N_i|^2)$ - we assume that $|N_i| \approx |N_j|$. $\Pi_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is not memory-optimal, but it has a shorter completion time $O(|N_i|)$. The length of the longest path in ISDG of Figure 5.2 is $|N_i|$. So, Π_1 is time-optimal.

The schedule of a loop in general and in particular in an embedded processing domain should be evaluated not only by its time but also by its memory requirements because embedded systems should operate in a real-time and its real-time performance should not be achieved at the expense of space. We will design two objective functions to evaluate a schedule from the perspective of both of time and space. In order to do that, we will include a storage vector into our objective function. In Figure 5.2 and 5.3, We justified the inclusion of a storage vector into our objective functions.

5.2 Legality Conditions and Objective Functions

Let D be a dependency matrix in which each column represents a dependency vector. A legal schedule, $\vec{\pi}$ should satisfy all dependency relations between computations.

$$\vec{\pi}\vec{d_i} \geq 1, \forall i \tag{5.1}$$

$$\vec{\pi}D \geq 1 \tag{5.2}$$

From Equation 5.1, we can characterize the region of feasible linear schedules for a given problem. By the definition of a storage vector, the delay of a storage vector is larger than or equal to the maximum delay of dependency vectors.

$$\vec{\pi}\vec{s} \geq \vec{\pi}D \tag{5.3}$$

When we choose a schedule and a storage vector, two objective functions will be used. The first objective function is

$$F_1 = \min(\max_{\forall i} \vec{\pi}\vec{d_i}).$$

If we can minimize the maximum delay of dependency vectors, it may be helpful to complete a problem in a shorter time. The second objective function is

$$F_2 = \min(|\vec{\pi}\vec{s} - \max_{\forall i} \vec{\pi}\vec{d_i}|).$$

When the delay of a storage vector is closer to the maximum delay of dependencies, memory will be reused more frequently and then less memory requirement will be guaranteed. For example, in Figure 5.2, according to our objective function F_1, schedules Π_1 and Π_2 have maximum delay 1 for a dependency vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, and a schedule Π_3 has a delay $|N_j|$. Obviously, our objective function F_1 prefers Π_1 and Π_2 to Π_3. Based on an objective function F_2, $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ will be a storage vector because it satisfies Equation 5.3, and has minimum value 0 for F_2.

5.3 Regions of Feasible Schedules and of Storage Vectors

Let $D_1 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix}$ be a dependency matrix. From the legality condition of a schedule, we can find the region of legal schedules. Let $\mathit{\Pi}_{D_1}$ be a region of legal schedules for D_1. From the Equation 5.1, $\vec{\pi} = (\pi_1, \pi_2)$ should satisfy all the dependencies.

$$(\pi_1, \pi_2) \begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix} \geq 1$$

Then, we have three inequalities.

$$\begin{aligned} \pi_1 &\geq 1 \\ \pi_1 - \pi_2 &\geq 1 \\ \pi_1 + 2\pi_2 &\geq 1 \end{aligned}$$

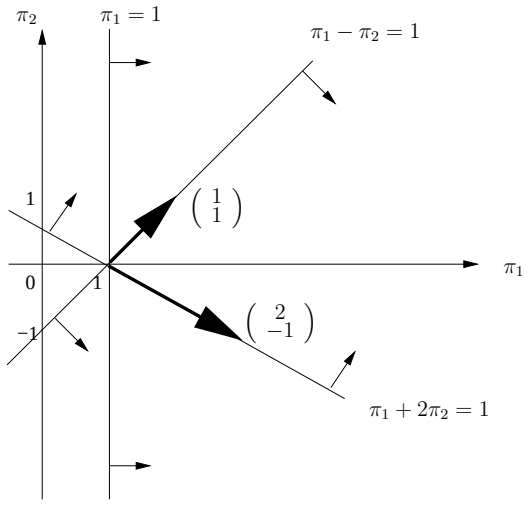


Figure 5.4: The region of feasible schedules, Π_{D_1}.

Figure 5.4 shows the region of legal schedules bounded by those three inequalities. This region is characterized by one corner and two extreme vectors [65]. In this example, a corner is in $(1, 0)$, and two extreme vectors are $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 2 \\ -1 \end{pmatrix}$. All the legal linear schedules for D_1 can be expressed by $\begin{pmatrix} \pi_1 \\ \pi_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \alpha \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \beta \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \alpha \geq 0, \beta \geq 0, \alpha, \beta \in R, \pi_1, \pi_2 \in Z$. In general, all the legal linear schedules can be expressed

by a following equation.

$$\begin{pmatrix} \pi_1 \\ \pi_2 \end{pmatrix} = \vec{c} + \alpha\vec{e_1} + \beta\vec{e_2}, \alpha, \beta \geq 0, \alpha, \beta \in R, \pi_1, \pi_2 \in Z \tag{5.4}$$

, where c is a corner and e_1 and e_2 are extreme vectors. From the region of feasible schedules, we can characterize a region of storage vectors for D_1 by a legality condition of a storage vector in Equation 5.3 with above two extreme vectors and the corner. From Equation 5.3 with two extreme vectors, $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\begin{pmatrix} 2 \\ -1 \end{pmatrix}$, and a corner $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, we have following inequalities.

$$\begin{aligned}
(1,1)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} &\geq (1,1)\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix} \\
(2,-1)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} &\geq (2,-1)\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix} \\
(1,0)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} &\geq (1,0)\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix}
\end{aligned}$$

Then,

$$s_1 + s_2 \geq \max(1,0,3) \tag{5.5}$$

$$2s_1 - s_2 \geq \max(2,3,0) \tag{5.6}$$

$$s_1 \geq \max(1,1,1). \tag{5.7}$$

Figure 5.5 shows the region of storage vectors. In this example $\vec{s} = (2,1)$ is on both of the boundary lines defined by inequalities in 5.5, and 5.6. When we use $\vec{s} = (2,1)$ in Equation 5.3, we can find feasible schedules for a storage vector, $\vec{s} = (2,1)$.

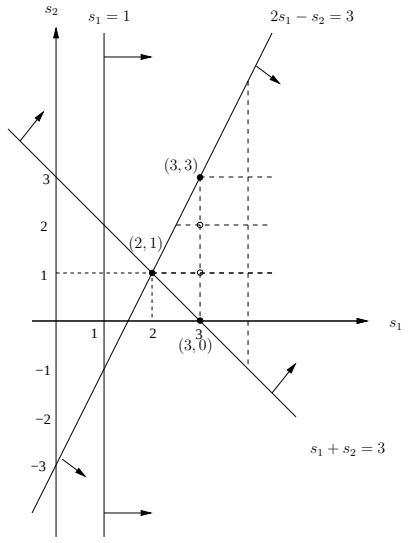


Figure 5.5: A region of storage vectors for D_1.

$$
\begin{aligned}
(\pi_1, \pi_2)\begin{pmatrix} 2 \\ 1 \end{pmatrix} &\geq (\pi_1, \pi_2)\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix} \\
2\pi_1 + \pi_2 &\geq \pi_1 \\
&\geq \pi_1 - \pi_2 \\
&\geq \pi_1 + 2\pi_2 \\
\Rightarrow \pi_1 + \pi_2 &\geq 0 \\
\pi_1 + 2\pi_2 &\geq 0 \\
\pi_1 - \pi_2 &\geq 0
\end{aligned}
$$

Then, the region of legal schedules for $\vec{s} = (2, 1)$ is bounded by two extreme vectors, $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 2 \\ -1 \end{pmatrix}$ as shown in Figure 5.6.

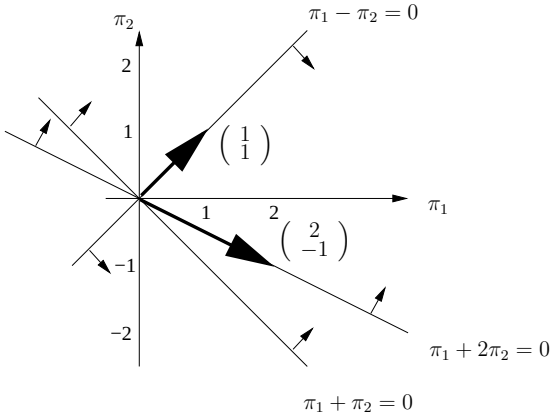


Figure 5.6: The region of legal schedules, $\Pi_{(2,1)}$ with $\vec{s} = (2, 1)$.

Let $\Pi_{\vec{s}}$ be a region of legal schedules under a storage vector, $\vec{s}$. In this example, $\Pi_{(2,1)}$ has same extreme vectors as $\mathit{\Pi}_{D_1}$, which means that $\Pi_{(2,1)}$ and $\mathit{\Pi}_{D_1}$ are exactly of the same shape. We will explain the meaning of the same shape of two regions from the perspective of an optimality of a storage vector.

5.4 Optimality of a Storage Vector

Definition 5.5 *In a two-dimensional iteration space, when two regions with different corners are bounded by same set of extreme vectors, it is said that the two regions have the same shape.*

When two different regions are of the same shape, it is possible to overlap exactly one region onto another by translation.

Definition 5.6 *When a storage vector $\vec{s}$ for a given problem D has its corresponding feasible schedule region $\Pi_{\vec{s}}$ that has a same shape as the region of feasible schedules Π_D for D, it is said that a storage vector $\vec{s}$ is optimal for D.*

In order to investigate the optimality of a storage vector, it is necessary to examine the relationship between various storage vectors and their corresponding $\Pi_{\vec{s}}$. In Figure 5.5, $\vec{s_1} = (3, 0)$ is on the line of $s_1 + s_2 = 3$ which comes from an extreme vector, $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ of Π_{D_1}, and below the line of $2s_1 - s_2 = 3$ which comes from an extreme vector, $\begin{pmatrix} 2 \\ -1 \end{pmatrix}$ of Π_{D_1}. From the storage legality condition of Equation 5.3 with $\vec{s_1} = (3, 0)$, we can find $\Pi_{(3,0)}$ as shown in Figure 5.7.

$$
\begin{aligned}
(\pi_1, \pi_2) \begin{pmatrix} 3 \\ 0 \end{pmatrix} &\geq (\pi_1, \pi_2) \begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix} \\
3\pi_1 &\geq \pi_1 \\
&\geq \pi_1 - \pi_2 \\
&\geq \pi_1 + 2\pi_2 \\
\Rightarrow \pi_1 &\geq 0 \\
2\pi_1 + \pi_2 &\geq 0 \\
2\pi_1 - 2\pi_2 &\geq 0
\end{aligned}
$$

Extreme vectors of $\Pi_{(3,0)}$ is $\left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix} \right\}$. $\Pi_{(3,0)}$ encloses Π_{D_1}. With $\vec{s_2} = (3, 3)$ that is on the line of $2s_1 - s_2 = 3$ and above the line of $s_1 + s_2 = 3$. In a similar way, we can find $\left\{ \begin{pmatrix} -1 \\ 2 \end{pmatrix}, \begin{pmatrix} 2 \\ -1 \end{pmatrix} \right\}$ extreme vectors of $\Pi_{(3,3)}$. $\Pi_{(3,3)}$ also encloses Π_{D_1}. A vector $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ is out of the region of storage vectors for D_1. When we choose $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ as a

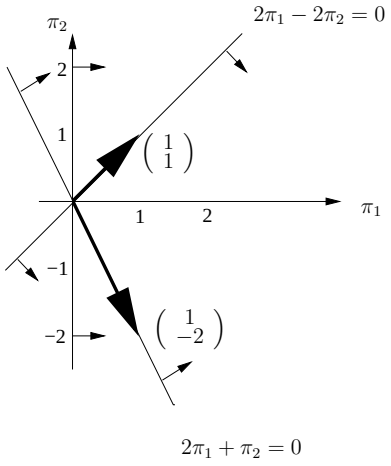


Figure 5.7: The region of legal schedules,$\Pi_{(3,0)}$ with $\vec{s_1} = (3, 0)$.

storage vector, the feasible region of its corresponding schedules is bounded by two extreme vectors $\left\{ \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}$. Figure 5.8 shows all the regions of schedules with different storage vectors. Both of $\vec{s_1} = (3, 0)$ and $\vec{s_2} = (3, 3)$ are legal storage vectors because their corresponding schedules, $\Pi_{(3,0)}$ and $\Pi_{(3,3)}$ enclose all the feasible linear schedules, Π_{D_1}, but obviously, $\vec{s} = (2, 1)$ is better than $\vec{s_1} = (3, 0)$ and $\vec{s_2} = (3, 3)$. $\Pi_{(3,0)}$ and $\Pi_{(3,3)}$ contain non-feasible schedules for a dependency matrix D_1, which means $\vec{s_1} = (3, 0)$, and $\vec{s_2} = (3, 3)$ are unnecessarily large in order for the corresponding schedules $\Pi_{\vec{s_1}}$ and $\Pi_{\vec{s_2}}$ to contain those non-feasible schedules. As you can see the shaded region in Figure 5.8, $\Pi_{(2,0)}$ does not enclose $\mathit{\Pi}_{D_1}$, which means that when we choose $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ as a storage vector, some feasible schedules can not satisfy the storage legality condition of Equation 5.3. However, it does not mean that there is no feasible schedules at all to satisfy Equation5.3. For a partial region of $\mathit{\Pi}_{D_1}$, $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ can be a storage vector if we allow the existence of some feasible schedules that does not satisfy Equation 5.3. We will explore a partial region of feasible

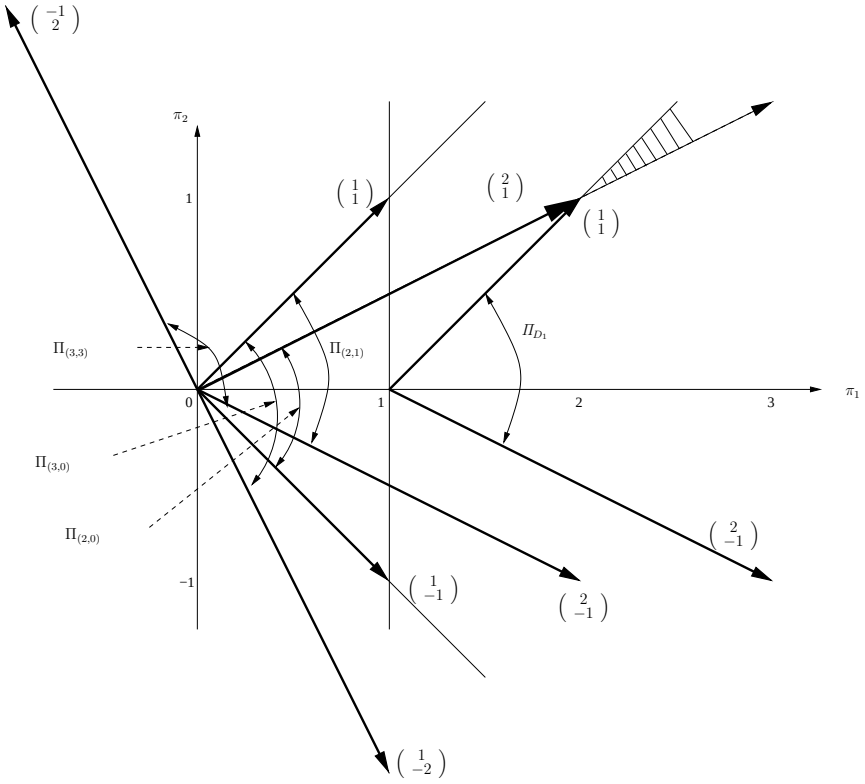


Figure 5.8: The regions of schedules with different storage vectors.

schedules to find a pair of a schedule and a storage vector that is favored by our objective function F_2. With a legality condition of a schedule and an objective function F_1, a corner $(1, 0)$ will be a good candidate for a schedule, because from the Equation 5.4 the delay of each dependence vector in D_1 are

$$(1 + \alpha + 2\beta, \alpha - \beta) \begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix} = (1 + \alpha + 2\beta, 1 + 3\beta, 1 + 3\alpha).$$

When $\alpha = \beta = 0$, the maximum delay is 1. It means a schedule $(1, 0)$ is optimal for F_1. Let us consider $(1, 0)$ as a schedule. From the perspective of our objective function F_2, $\vec{s} = (2, 1)$ is a preferred storage vector under the schedule $\vec{\pi} = (1, 0)$ because

$$\begin{aligned} \left| (1,0) \begin{pmatrix} 2 \\ 1 \end{pmatrix} - \max(1,1,1) \right| &= 1 \\ \left| (1,0) \begin{pmatrix} 3 \\ 0 \end{pmatrix} - \max(1,1,1) \right| &= 2 \\ \left| (1,0) \begin{pmatrix} 3 \\ 3 \end{pmatrix} - \max(1,1,1) \right| &= 2. \end{aligned}$$

From the observation of the above three specific feasible storage vectors and one partially feasible storage vector, we can conclude that if a corner of a region of a storage vector happens to be in integer lattice, the corner is always a preferred storage vector. If it is not the case, the nearest integer lattice might be preferred.

5.5 A More General Example

Let $D_2 = \begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix}$. From the legality condition of a schedule of Equation 5.1, we have following inequalities.

$$\begin{aligned} \vec{\pi} D_2 &\geq 1 \\ \pi_1 &\geq 1 \\ \pi_1 - \pi_2 &\geq 1 \\ 2\pi_1 + \pi_2 &\geq 1. \end{aligned}$$

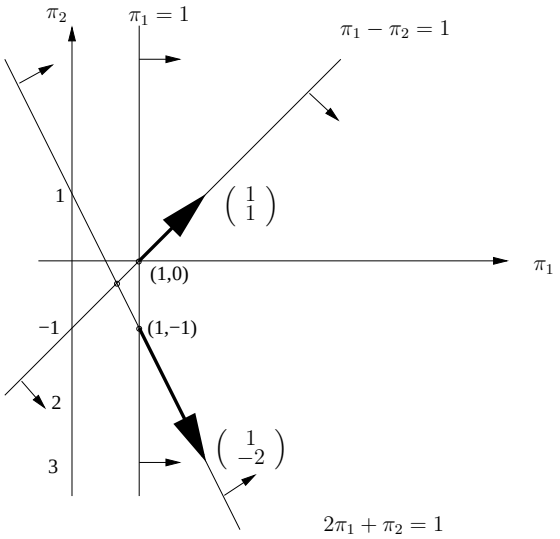


Figure 5.9: The region of feasible schedules, Π_{D_2} for D_2.

Figure 5.9 shows the region of feasible schedules, Π_{D_2}. Π_{D_2} is characterized by two corners $(1, 0), (1, -1)$ and two extreme vectors, $\left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix} \right\}$. Π_{D_2} consists of two

subregions, $\Pi_{D_2}(1,0), \Pi_{D_2}(1,-1)$ which are not necessarily disjoint. Figure 5.10 shows those two subregions. $\Pi_{D_2}(1,0)$ is a subregion whose corner is $(1,0)$, and $\Pi_{D_2}(1,-1)$

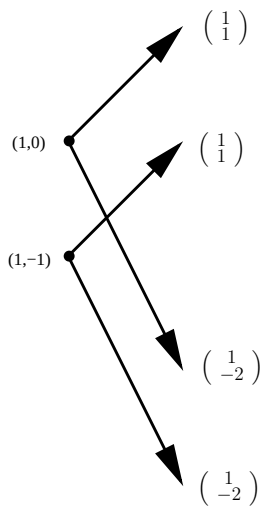


Figure 5.10: Two subregions of Π_{D_2}.

is a subregion whose corner is $(1,-1)$. Both of them are bounded by the same extreme vectors. $\Pi_{D_2}(1,0)$ is to be characterized by three vectors, $\left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix} \right\}$, and $\Pi_{D_2}(1,-1)$ by $\left\{ \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix} \right\}$. The first element is a corner, and the last two are extreme vectors. From the legality condition of a storage vector, we can find the region of storage vectors for each subregion of feasible Π_{D_2}. In this example, $\Pi_{D_2}(1,0)$ and $\Pi_{D_2}(1,-1)$ have same region of storage vectors. Figure 5.11 shows the region of storage vectors.

From Equation 5.3 with two extreme vectors,

$$(1,1)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} \geq (1,1)\begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix}$$

$$(1,-2)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} \geq (1,-2)\begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix}$$

$$\begin{aligned} s_1 + s_2 &\geq \max(1,0,3) \\ s_1 - 2s_2 &\geq \max(1,3,0) \\ \Rightarrow s_1 + s_2 &\geq 3 \\ s_1 - 2s_2 &\geq 3. \end{aligned}$$

With a corner $(1,0)$ for $\mathit{\Pi}_{D_2}(1,0)$,

$$\begin{aligned} (1,0)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} &\geq (1,0)\begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix} \\ s_1 &\geq \max(1,1,2) \\ \Rightarrow s_1 &\geq 2. \end{aligned}$$

With a corner $(1,-1)$ for $\mathit{\Pi}_{D_2}(1,-1)$,

$$\begin{aligned} (1,-1)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} &\geq (1,-1)\begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix} \\ s_1 - s_2 &\geq \max(1,2,1) \\ \Rightarrow s_1 - s_2 &\geq 2. \end{aligned}$$

$\vec{s_1} = (3,0)$ is on the both lines of $s_1 - 2s_2 = 3$ from an extreme vector $\begin{pmatrix} 1 \\ -2 \end{pmatrix}$, and $s_1 + s_2 = 3$ from an extreme vector $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$. So $\Pi_{(3,0)}$ is of the same shape as $\mathit{\Pi}_{D_2}$,

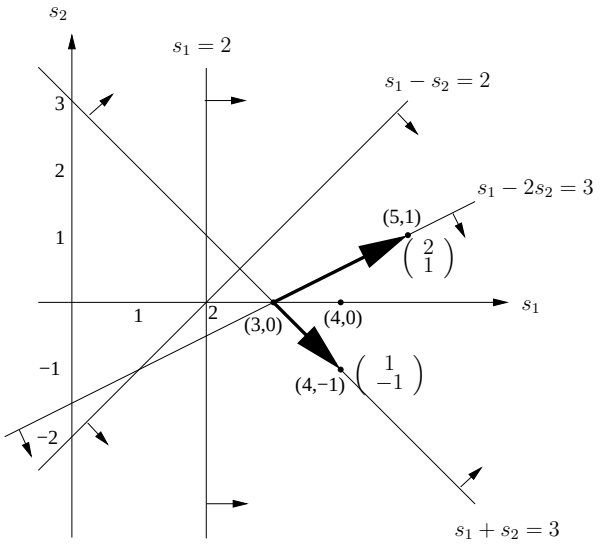


Figure 5.11: Storage vectors for D_2.

which means that $\vec{s_1} = (3, 0)$ is just as large as it is supposed to be in order to enclose Π_{D_2}. In that sense, $\vec{s_1} = (3, 0)$ is an optimal storage vector for D_2. Corners of Π_{D_2} are good candidates for a objective function F_1. A schedule $\vec{\pi_1} = (1, 0)$ has a maximum delay 2 for a dependency vector $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$, and a schedule $\vec{\pi_2} = (1, -1)$ has also a maximum delay 2 for a dependency vector $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$. With an optimal storage vector $\vec{s_1} = (3, 0)$, we can evaluate a pair $(\vec{\pi}, \vec{s})$ of a schedule and a storage vector based on objective function F_2. For the pair $\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \end{pmatrix}\right)$,

$$\left|(1, 0)\begin{pmatrix} 3 \\ 0 \end{pmatrix} - 2\right| = 1.$$

For the pair $\left(\begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \end{pmatrix}\right)$,

$$\left|(1,-1)\begin{pmatrix} 3 \\ 0 \end{pmatrix} - 2\right| = 1.$$

Definition 5.7 *When a storage vector $\vec{s}$ is not optimal for a given problem D, if there exist some feasible schedules $\vec{\pi}$ in Π_D such that those schedules satisfy a legality condition of a storage vector $\vec{s}$ and a pair $(\vec{\pi}, \vec{s})$ has a value 0 for F_2, the pair $(\vec{\pi}, \vec{s})$ is called specifically optimal for F_2.*

When the delay of a storage vector is same as the maximum delay of dependency vectors under a certain schedule $\vec{\pi}$ i.e.,$(\vec{\pi}\vec{s} = \max_{\forall i} \vec{\pi}\vec{d_i})$, we may think that under that schedule a storage vector $\vec{s}$ is specifically optimal for that schedule $\vec{\pi}$ because by the definition of a storage vector the delay of storage vector can not be shorter than the maximum delay of dependency vectors. In the above example, F_2 has a value 1, which means that $(\vec{\pi_1}, \vec{s_1})$ and $(\vec{\pi_2}, \vec{s_1})$ are not specifically optimal from the perspective of F_2.

Up to this point, for a given problem we can find the region of feasible schedules, Π, and characterize the region of corresponding storage vectors with (a) corner(s) and extreme vectors of Π. We can evaluate a pair of a schedule and a storage vector by objective F_2. We may have a question at this point like "Is it possible to find specifically optimal pairs?". In order to find an answer to this question, we try to generate several possible pairs. We can partition the region of feasible schedules, Π into several subregions. Figure 5.12 shows those subregions.

Obviously, all subregions of Π_{D_2} are feasible schedules for D_2. By picking up two internal vectors arbitrarily, we can generate feasible subregions. Let $\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ be two extreme vectors for a subregion. We can find the region of storage vectors for this

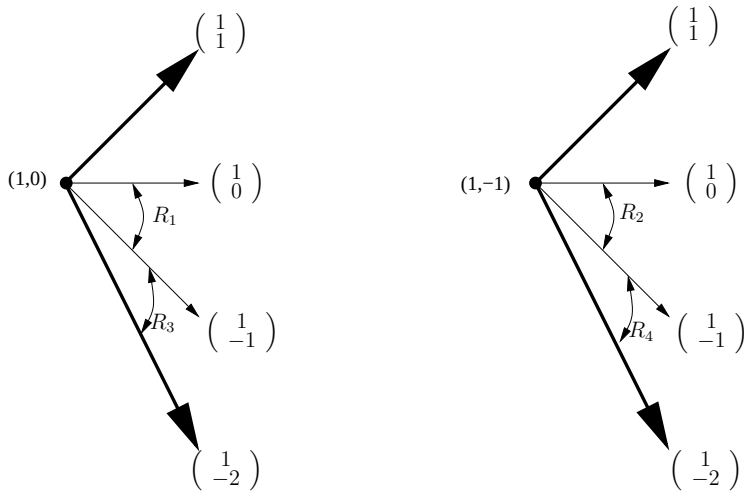


Figure 5.12: Partitions of each subregions of Π_{D_2}.

scheduling subregion. From the legality condition of a storage vector,

$$\begin{aligned}
(1,0)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} &\geq (1,0)\begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix} \\
s_1 &\geq \max(1,1,2) \\
(1,-1)\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} &\geq (1,-1)\begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix} \\
s_1 - s_2 &\geq \max(1,2,1).
\end{aligned}$$

Coincidentally, two corners of $\mathit{\Pi}_{D_2}$ are same as extreme vectors in this example. Figure 5.13 shows the region of storage vectors. $\vec{s_3} = (2,0)$ is a corner of the region of storage vectors. For the two subregions, $R_1 = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}$, and $R_2 = \left\{ \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}$, $\vec{s_3} = (2,0)$ is an optimal storage vector for R_1 and R_2 because $\Pi_{\vec{s_3}}$ is bounded by extreme vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$, which means that $\Pi_{\vec{s_3}}$ is of

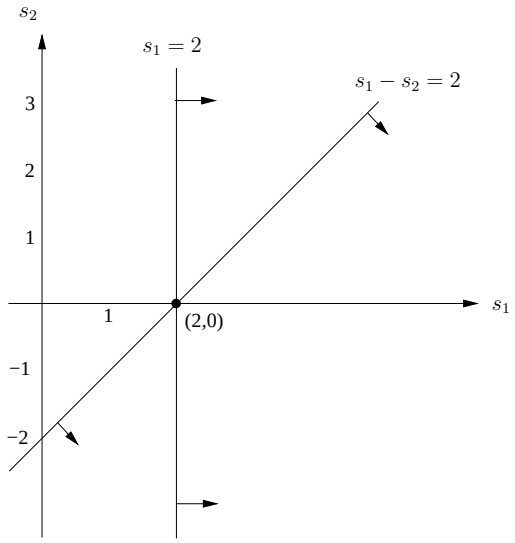


Figure 5.13: Storage vectors for the region of schedules bounded by $(1,0),(1,-1)$.

the same shape of the two subregions R_1 and R_2. However, $\vec{s_3} = (2,0)$ is not an optimal storage vector for Π_{D_2} as we can see in Figure 5.11, in which $(2,0)$ is out of the region of storage vectors for D_2. Corners $(1,0)$ and $(1,-1)$ are good candidate schedules for F_1. We can evaluate the pair $\left\{\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}2\\0\end{pmatrix}\right\}$, $\left\{\begin{pmatrix}1\\-1\end{pmatrix},\begin{pmatrix}2\\0\end{pmatrix}\right\}$ with F_2.

$$\begin{aligned}\left|(1,0)\begin{pmatrix}2\\0\end{pmatrix} - \max\left((1,0)\begin{pmatrix}1&1&2\\0&-1&1\end{pmatrix}\right)\right| &= 0\\ \left|(1,-1)\begin{pmatrix}2\\0\end{pmatrix} - \max\left((1,-1)\begin{pmatrix}1&1&2\\0&-1&1\end{pmatrix}\right)\right| &= 0.\end{aligned}$$

The pairs $\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}2\\0\end{pmatrix}\right)$ and $\left(\begin{pmatrix}1\\-1\end{pmatrix},\begin{pmatrix}2\\0\end{pmatrix}\right)$ are specifically optimal for R_1 and R_2 respectively. Let $\begin{pmatrix}1\\-1\end{pmatrix}$ and $\begin{pmatrix}1\\-2\end{pmatrix}$ be two extreme vectors of another subregion R_3 and R_4. Then, the region of corresponding storage vectors is shown in Figure 5.14.

$(3,0)$ and $(2,-1)$ are two integer points close to a corner $(2,-1/2)$. We already know that

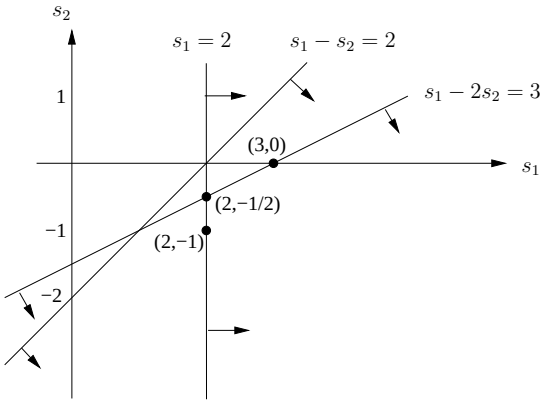


Figure 5.14: Storage vectors for the region of schedules bounded by $(1,-1),(1,-2)$.

a storage vector $(3,0)$ can not specifically optimal. In the case of $\vec{s_4} = (2,-1)$, the pair $\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}2\\-1\end{pmatrix}\right)$ is specifically optimal with F_2 but the pair $\left(\begin{pmatrix}1\\-1\end{pmatrix},\begin{pmatrix}2\\-1\end{pmatrix}\right)$ is not. From arbitrarily chosen four subregions R_1, R_2, R_3, R_4, we have found 3 three specifically optimal pairs. Figure 5.15 summarizes our approach to find the pairs.

5.6 Finding a Schedule for a Given Storage Vector

When a candidate storage vector $\vec{s}$ is given, we can determine whether the given vector $\vec{s}$ is valid or not. If a given vector $\vec{s}$ is valid, we could find the best schedule for the vector $\vec{s}$. Let us take D_2 of the previous section be a given dependence matrix. For D_2, we could ask a question like "Is $\vec{s} = (1,0)$ valid?". In order to answer this question, we need to find a feasible scheduling region, $\Pi_{\vec{s}}$ for $\vec{s}$.

There might be three possibilities; The regions of $\Pi_{\vec{s}}$ and $\mathit{\Pi}_{D_2}$ are disjoint, partially overlapped or exactly overlapped from the perspective of extreme vectors that define each

Procedure Find_Main(D)
D : a dependence matrix
{
Find a region $\mathit{\Pi}_D$ of feasible schedules from the legality condition of a schedule;
return Find_Pair($\mathit{\Pi}_D$)
}

Procedure Find_Pair(Π)
Π : a region of feasible schedules
{
Find a region $\mathcal{S}$ of storage vectors from the legality condition of a storage vector with (a) corner(s) and extreme vectors of Π;
Find a corner of $\mathcal{S}$ **do**
  if it is not in integer point
    find nearest integer point(s);
  endif
enddo
Choose (a) corner(s) of Π as a schedule;
Choose (a) corner(s) of $\mathcal{S}$ as a storage vector;
if a pair $(\vec{\pi}, \vec{s})$ has 0 for F_2
  return $(\vec{\pi}, \vec{s})$;
else if Π is divisible into subregions
  divide Π into subregions;
  for each subregion $R \in \Pi$ **do**
    Find_Pair(R);
  enddo
else
  return
endif
if there is no pair with 0 for F_2
  choose the pair with the smallest value for F_2;
endif

return the best pair found;
}

Figure 5.15: Our approach to find specifically optimal pairs.

region of $\Pi_{\vec{s}}$ and $\mathit{\Pi}_{D_2}$. From the legality condition of a storage vector with $\vec{s_5} = (1, 0)$, we can find $\Pi_{\vec{s}}$ in a similar way of the previous section. Figure 5.16 shows the region of corresponding schedule for $\vec{s_5} = (1, 0)$. When we position the corner of $\Pi_{(1,0)}$ at the

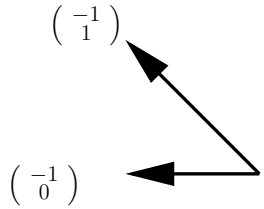


Figure 5.16: $\Pi_{(1,0)}$.

same corner of $\mathit{\Pi}_{D_2}$, they are disjoint, which means that when $\vec{s_5} = (1, 0)$ is selected for a storage vector for a dependency matrix D_2, there is no feasible schedules exist for a given problem D_2. When $\vec{s_3} = (2, 0)$ is given, we can tell $\Pi_{(2,0)}$, which was already computed in the previous section, is partially overlapped with Π_{D_2}. In this case, $\vec{s_3} = (2, 0)$ is a valid storage vector only for schedules in $\Pi_{(2,0)}$. Figure 5.17 shows $\Pi_{(2,0)}$. For all the schedules

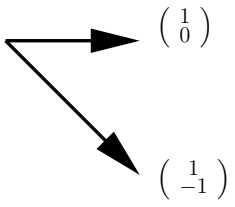


Figure 5.17: $\Pi_{(2,0)}$.

that belong to $\Pi_{(2,0)}$, $\vec{s_3} = (2, 0)$ is valid, but for the other schedules, except (a) corner(s), that belong to Π_{D_2} but do not belong to $\Pi_{(2,0)}$, $\vec{s_3} = (2, 0)$ is not valid.

5.7 Finding a Storage Vector from Dependence Vectors

From the legality condition for a storage vector, we can directly find a legal storage vector for any legal linear schedule for a set of dependence vectors. We limit the discussion here to two-level nested loops. Note that these results hold true for any n-level nested loop in which there is a subset of n dependence vectors which are extreme vectors. This is always the case for $n = 2$.

For the rest of this discussion, we assume a two-level nested loop. Let the dependence matrix D be $(\vec{d_1}, \vec{d_2}, \cdots, \vec{d_m})$. Let $\vec{r_1}$ and $\vec{r_2}$ be the two extreme vectors of the dependence matrix D. All the dependence vectors in D can be specified as a non-negative linear combination of the two extreme vectors $\vec{r_1}, \vec{r_2}$.

$$\vec{d_i} = \alpha_i \vec{r_1} + \beta_i \vec{r_2},\ \alpha_i, \beta_i \geq 0, \alpha_i, \beta_i \in R, 1 \leq i \leq m. \tag{5.8}$$

Lemma 5.1 *Let* $\alpha_{max} = \max_i \alpha_i$ *and* $\beta_{max} = \max_i \beta_i$. *Let* $\vec{s}_{max} = \lceil \alpha_{max} \rceil \vec{r_1} + \lceil \beta_{max} \rceil \vec{r_2}$. *Then,* $\vec{s}_{max}$ *is a legal storage vector for any legal linear schedule* $\vec{\pi}$.

(Proof) Let $\delta_1 = \vec{\pi}\vec{r_1}$ and $\delta_2 = \vec{\pi}\vec{r_2}$ for some schedule vector $\vec{\pi}$. From Equation 5.1, $\delta_1 \geq 1$ and $\delta_2 \geq 1$. From the legality condition for a storage vector in Equation 5.3 and Equation 5.8, we have

$$\begin{aligned} \vec{\pi}\vec{s} &\geq \vec{\pi}\vec{d_i},\ \forall i \\ &= \alpha_i \delta_1 + \beta_i \delta_2 \\ &\geq 1 \end{aligned}$$

$$\begin{aligned} \Rightarrow \vec{\pi}\vec{s}_{max} &= \lceil \alpha_{max} \rceil \delta_1 + \lceil \beta_{max} \rceil \delta_2 \\ &\geq \alpha_i \delta_1 + \beta_i \delta_2, \ \forall i. \end{aligned}$$

So, $\vec{s}_{max}$ is a valid storage vector for any schedule $\vec{\pi}$.

Examples Let us consider the dependence matrix $D_1 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix}$ as in Section 5.3, the two extreme vectors are $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$. All the dependence vectors can be written as non-negative linear combination of the extreme vectors as follows.

$$\begin{aligned} \begin{pmatrix} 1 \\ -1 \end{pmatrix} &= 1 \begin{pmatrix} 1 \\ -1 \end{pmatrix} + 0 \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ \begin{pmatrix} 1 \\ 2 \end{pmatrix} &= 0 \begin{pmatrix} 1 \\ -1 \end{pmatrix} + 1 \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} &= \frac{2}{3} \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \frac{1}{3} \begin{pmatrix} 1 \\ 2 \end{pmatrix}. \end{aligned}$$

So, $\lceil \alpha_{max} \rceil = 1, \lceil \beta_{max} \rceil = 1$. Then,

$$\begin{aligned} \vec{s}_{max} &= 1 \begin{pmatrix} 1 \\ -1 \end{pmatrix} + 1 \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ &= \begin{pmatrix} 2 \\ 1 \end{pmatrix}. \end{aligned}$$

$\vec{s}_{max} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ is same as the corner of the region of feasible storage vectors that was found in Section 5.3.

Consider a different dependence matrix $D_2 = \begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & 1 \end{pmatrix}$ as in Section 5.3; $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ and $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ are the extreme vectors. We find $\lceil \alpha_{max} \rceil = 1, \lceil \beta_{max} \rceil = 1$. The vector $\vec{s}_{max}$ is

$\begin{pmatrix} 3 \\ 0 \end{pmatrix} = 1\begin{pmatrix} 1 \\ -1 \end{pmatrix} + 1\begin{pmatrix} 2 \\ 1 \end{pmatrix}$. Again, $\vec{s}_{max} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$ is the same as the corner of the region of feasible storage vectors for D_2.

5.8 UOV Algorithm

Strout [74] shows that a difference vector $\vec{v} = (\vec{c_2} - \vec{c_1})$ is a UOV if it is possible that all of the value dependences have been traversed at least once to reach $\vec{c_2}$from $\vec{c_1}$. In order to find a UOV, his algorithm keeps $PATHSET$ in each iteration point while traversing iteration space. $PATHSET$ will contain dependence vectors that have been traversed from a starting point to the current point. If $PATHSET$ of an iteration point contain all dependence vectors, the difference vector of the current point and a starting point is a UOV. He uses priority queue hoping find a UOV quickly. In our algorithm, we do not use priority queue and do not keep $PATHSET$ in each iteration point. Instead, we expand an iteration space from an arbitrary starting iteration point - for convenience of computing a UOV, an origin $\vec{0}$ is used in our algorithm. We call this iteration space a partially expanded ISDG or a partial ISDG. Our algorithm expands an iteration space level by level from the starting iteration point by adding dependence vectors.

Lemma 5.2 *When $|D| = k$, if k immediate predecessors of $\vec{c}$ belong to a partial ISDG, $\vec{c}$ is a UOV.*

Proof: Given the manner in which we generate a partial ISDG, it follows that all the k immediate predecessors $\vec{c}_0, \cdots, \vec{c}_{k-1}$ are reachable from the starting point $\vec{0}$, which means that there are k different paths from $\vec{0}$ to $\vec{c}$; $P_0 = \vec{0} \rightsquigarrow \vec{c}_0 \rightarrow \vec{c}, P_1 = \vec{0} \rightsquigarrow \vec{c}_1 \rightarrow \vec{c}, \cdots, P_{k-1} = \vec{0} \rightsquigarrow \vec{c}_{k-1} \rightarrow \vec{c}$. In each different path, at least one dependence vector is guaranteed to be traversed. Each path $P_i, 0 \leq i < k$ guarantees a different dependence vector $(\vec{c} - \vec{c_i})$ to be traversed. So, $\vec{c}$ is a UOV.

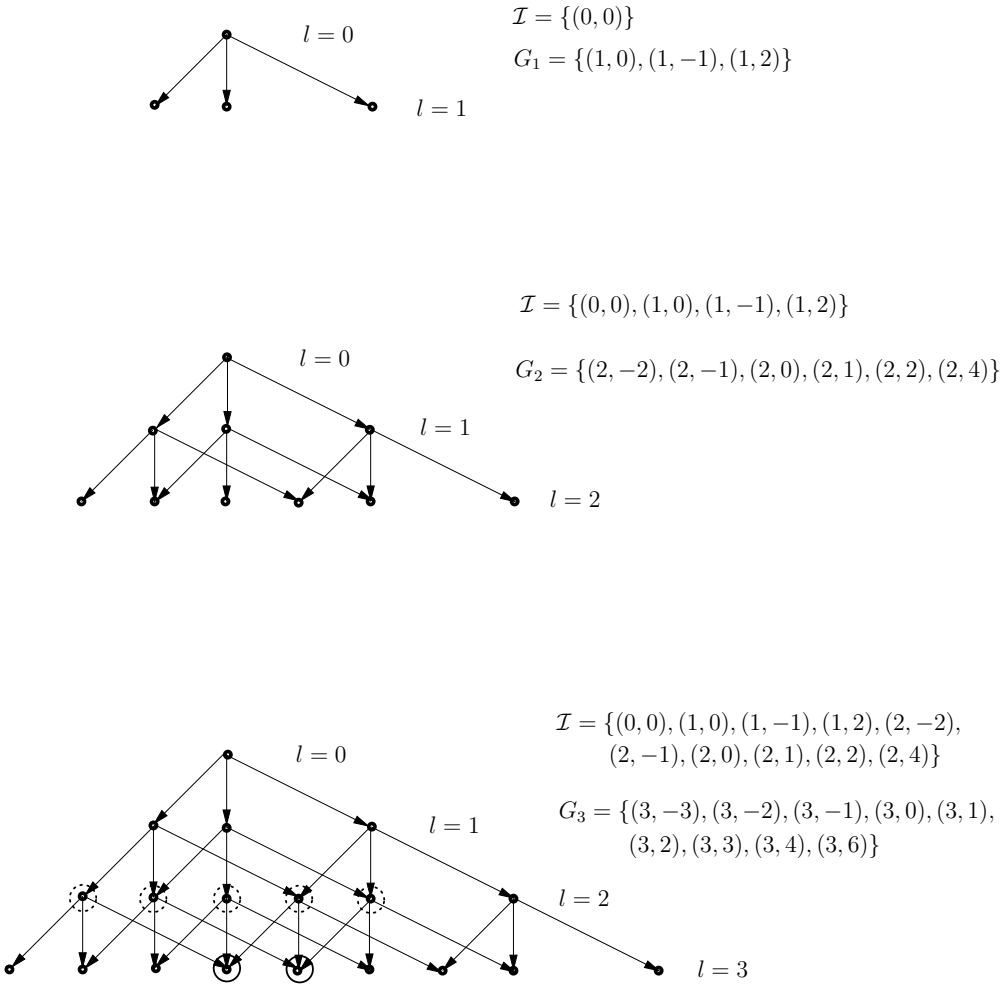


Figure 5.18: How to find a UOV.

Figure 5.18 shows how our algorithm works. At level 0 there is only one iteration point. The iteration points at level 1 can be generated by adding dependence vectors to the iteration point at level 0. All the iteration points at level i will be generated by adding dependence vectors to the points at level $(i-1)$. In this way, we generate a partial ISDG. After expanding all the iteration points at the current level, we check if there is an iteration point at the current level, all of whose k immediate predecessors belong to the partial ISDG. If there is such an iteration point $\vec{c}$, then $(\vec{c}-\vec{0})$ is a UOV.

5.9 Experimental Results

We experiment our UOV algorithm with several scenarios. We generate legal dependence vectors, and then apply our UOV algorithm. We repeat our experiment 100 times in each scenario. Tables 5.1 and 5.2 show the results. In Table 5.1, we compare the size of an UOV that our algorithm found with the average size of dependence vectors. The average size of dependence vectors is defined as follows. When a dependence matrix

$$D = \begin{pmatrix} d_{11} & d_{12} & \dots & d_{1k} \\ d_{21} & d_{22} & \dots & d_{2k} \\ \vdots & \vdots & \cdots & \vdots \\ d_{n1} & d_{n2} & \cdots & d_{nk} \end{pmatrix},$$

the average size of D is defined as $\frac{\sum_{j=1}^{k}\sum_{i=1}^{n}|d_{ij}|}{k}$.

The first column is the number of dependence vectors. The second column is the range that each element of a dependence vector can take. For example, when the range is 3, the elements of a dependence vector can have a value between -3 and 3. Columns 3 through 8 show the number of dimensions. We refer to the ratio of the the size of the UOV to the average size of dependence vectors as simply the ratio. From the results in Table 5.1, it is difficult to find some regularities that could give us useful interpretation. When the number

```
Procedure Find UOV(D)
D : a dependence matrix
{
I ← {(0, ··· , 0)}; flag ← false; UOV ← {}; G ← I          /* Initialization */
while (flag == false) do
  G' ← {};
  for each g ∈ G do
    for each d ∈ D do
      e ← g + d;
      if (e ∉ G')
        G' ← G' ∪ {e};
      endif
    enddo
  enddo
  G ← G';
  for each g ∈ G do
    uovflag ← 0;
    for each d ∈ D do
      cand ← g − d;
      if (cand ∈ I)
        uovflag ← uovflag + 1;
      endif
    enddo
    if (uovflag = |D|)
      UOV ← UOV ∪ {g};
      flag ← true;
    endif
  enddo
  if (!flag)
    I ← I ∪ G;
  endif
endwhile
return UOV;
}
```

Figure 5.19: A UOV algorithm.

of dependence vectors is 6, the range is 5, and dimension is 4, the largest ratio is 3.37, which means that the size of a UOV is 3.37 times the average size of dependence vectors. The smallest ratio is 1.42 when the number of dependence vectors is 6, the range is 2, and dimension is 2.

Table 5.2 shows the execution time taken in seconds to find UOVs. Because the size of a partial ISDG grows exponentially with an increasing level, our UOV algorithm has an exponentially time complexity. We implemented our algorithm in Java on a sun workstation. Table 5.2 shows that in dimensions greater than 3, the number of dependence vectors has a huge impact on an execution time. For example, there is a big gap of an execution time between 5 dependence vectors and 6 dependence vectors in dimension 4, 5, 6, and 7. When the number of dependence vectors is 5, the range is 5, and a dimension is 4, the execution time is 70.607 seconds. On the contrary, when the number of dependence vectors is 6, the range is 2, and a dimension is 4, the execution time is 667.787 seconds. In a 5-dimensional space, the corresponding execution times are 71.072 seconds and 1203.167 seconds. We observe similar big gaps in higher dimensions in Table 5.2.

5.10 Chapter Summary

In this chapter, we have developed a framework for studying the trade-off between a schedule and storage requirements. We developed methods to compute the region of feasible schedules for a given storage vector. In previous work, Strout et al. [74] have developed an algorithm for computing the universal occupancy vector which is the storage vector that is legal for any schedule of the iterations. By this, Strout et al. [74] mean any topological ordering of the nodes of an iteration space dependence graph (ISDG). Our work is applicable to wavefront schedules of nested loops.

Table 5.1: The result of UOV algorithm with 100 iterations. (Average Size).

# of Dep.	range	Dimension					
		2	3	4	5	6	7
3	2	1.90	2.16	2.10	2.01	2.01	2.03
	3	2.12	2.15	2.03	2.02	1.89	1.93
	4	2.25	2.20	2.10	1.90	1.97	1.84
	5	2.40	2.17	2.10	1.97	1.91	1.86
4	2	1.61	2.39	2.53	2.49	2.40	2.47
	3	2.14	2.53	2.45	2.45	2.32	2.29
	4	2.51	2.65	2.37	2.38	2.40	2.35
	5	2.68	2.72	2.45	2.27	2.30	2.14
5	2	1.55	2.27	2.85	2.86	2.87	2.91
	3	1.94	2.72	2.94	2.87	2.73	2.68
	4	2.27	2.95	3.01	2.80	2.70	2.71
	5	2.45	3.25	2.88	2.79	2.76	2.66
6	2	1.42	2.16	2.64	3.30	3.30	3.26
	3	1.90	2.60	3.27	3.28	3.07	3.06
	4	2.10	3.07	3.30	3.18	3.08	2.94
	5	2.46	3.28	3.37	3.23	3.15	3.03

Table 5.2: The result of UOV algorithm with 100 iterations. (Execution Time).

# of Dep.	range	Dimension					
		2	3	4	5	6	7
3	2	0.308	0.372	0.345	0.371	0.409	0.380
	3	0.326	0.371	0.356	0.374	0.414	0.382
	4	0.352	0.378	0.354	0.371	0.408	0.379
	5	0.359	0.372	0.356	0.364	0.409	0.375
4	2	0.857	3.952	4.368	4.515	5.145	4.595
	3	2.403	4.656	4.461	4.491	5.132	4.574
	4	3.071	4.759	4.468	4.491	5.143	5.198
	5	3.677	4.806	4.454	4.484	5.132	4.540
5	2	1.396	25.248	62.671	70.914	80.424	81.244
	3	4.868	58.247	70.336	71.107	79.410	71.623
	4	8.312	65.135	70.269	71.139	79.998	71.154
	5	15.685	72.915	70.607	71.072	80.550	80.250
6	2	2.352	65.966	667.787	1203.167	1282.335	1291.654
	3	8.983	371.780	1081.227	1286.509	1288.086	1270.653
	4	18.138	758.323	1250.806	1282.241	1281.018	1269.210
	5	35.554	832.975	1270.128	1280.075	1280.769	1267.161

CHAPTER 6

TILING FOR IMPROVING MEMORY PERFORMANCE

Tiling (or loop blocking) has been one of the most effective techniques for enhancing locality is perfectly nested loops [15, 23, 80, 81, 40, 64, 65, 67, 68, 43]. Unimodular loop transformations such as skewing are necessary in some cases to render tiling legal. Irigoin and Triolet [40] developed a sufficient condition for tiling. It was conjectured by Ramanujam and Sadayappan [64, 65, 67] that this sufficient condition becomes necessary for "large enough" tiles, but no precise characterization is known.

A tile is an atomic unit in which all iteration points will be executed collectively before the execution thread leaves the tile. Tiling changes the order in which iteration points are executed [79, 81]. It does not eliminate or add any iteration point. So, the size of a tiled space is same as the size of an original space. Even though several iteration points may be mapped into the same tile, tiling is a one-to-one mapping.

A tile is specified by a set of vectors, which can be expressed by a tiling matrix B.

$$B = (\vec{b_1}\vec{b_2}\cdots\vec{b_n}), \vec{s_i} = (b_{1i}, b_{2i}, \cdots, b_{ni})^T, 1 \leq i \leq n$$

An iteration point $\vec{c} = (i_1, i_2, \cdots, i_n)^T$ in n-dimension space is mapped to the corresponding point $\vec{c'}$ in $2n$-dimension tiled space.

$$B : \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{pmatrix} \xrightarrow{tiled} \begin{pmatrix} i'_1 \\ i'_2 \\ \vdots \\ i'_n \\ i'_{n+1} \\ \vdots \\ i'_{2n} \end{pmatrix}.$$

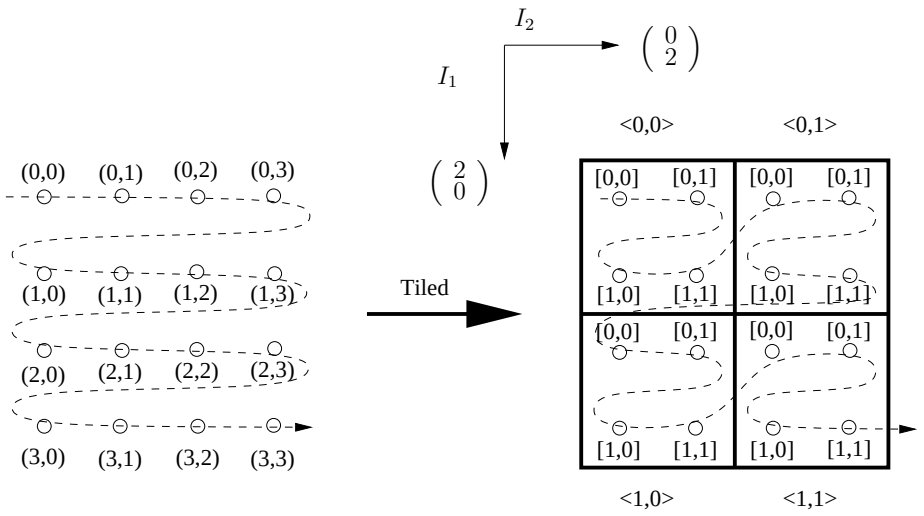
Let $\vec{t}$ be $(i'_1, i'_2, \cdots, i'_n)^T$ and $\vec{l}$ be $(i'_{n+1}, i'_{n+2}, \cdots, i'_{2n})^T$.

$$B : \vec{c} \xrightarrow{tiled} \begin{pmatrix} \vec{t} \\ \vec{l} \end{pmatrix}$$

$\vec{t}$ is an inter-tile coordinate, and $\vec{l}$ is an intra-tile coordinate.

Figure 6.1 shows an original space and the tiled space. In Figure 6.1, the arrows show the execution orders. In the original space, an iteration point $(0, 2)^T$ is executed immediately after an iteration point $(0, 1)^T$. However, in the tiled space iteration points $(1, 0)^T$ and $(1, 1)^T$ will be executed immediately after an iteration point $(0, 1)^T$, and an iteration point $(0, 2)^T$ will be executed immediately after an iteration point $(1, 1)^T$ which is supposed to be executed after $(0, 2)^T$ in the original space. Because the execution order of iteration points in the tiled space is different from the execution order in the original space, tiling should be applied carefully not to violate dependence relations in the original space.

In Figure 6.1, the tiles are specified by the matrix $B_1 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$. The absolute value of the determinant of B is equal to the number of iteration points in each tile. The determinant of B_1 is 4. Each tile in Figure 6.1 contains four iteration points. The mapping



$\mathcal{I} = \{(i,j) | 0 \leq i \leq 3, 0 \leq j \leq 3\}$

(a) A original space $\mathcal{I}$

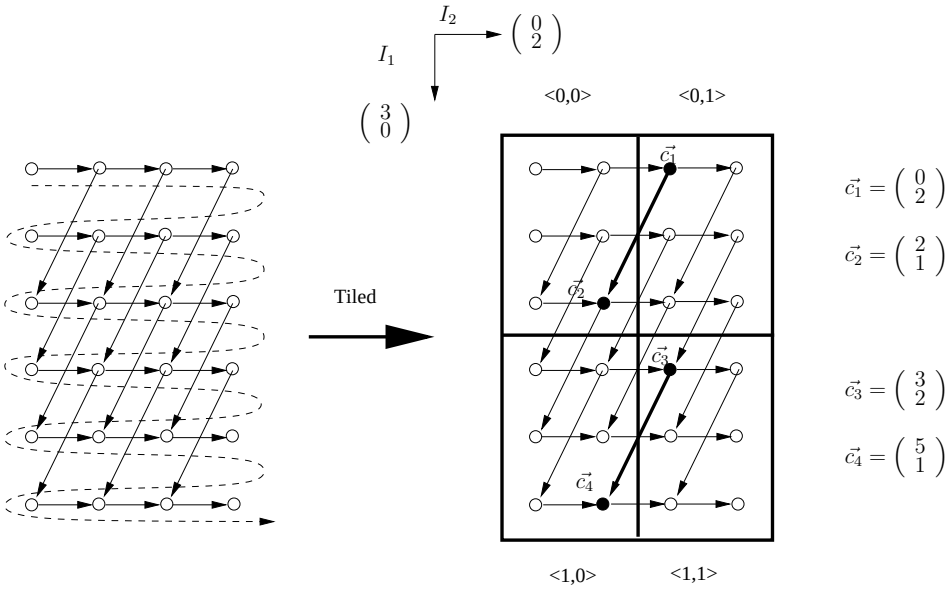
$\mathcal{I}_{tiled} = \{(t_i, t_j, l_i.l_j) | 0 \leq t_i \leq 1, 0 \leq t_j \leq 1, 0 \leq l_i \leq 1, 0 \leq l_j \leq 1\}$

(b) A tiled space $\mathcal{I}_{tiled}$

Figure 6.1: Tiled space.

of four iteration points in the tile $< 0, 0 >$ is as follows.

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \rightarrow (\mathbf{0}, \mathbf{0}, 0, 0)^T$$
$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \rightarrow (\mathbf{0}, \mathbf{0}, 0, 1)^T$$
$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \rightarrow (\mathbf{0}, \mathbf{0}, 1, 0)^T$$
$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \rightarrow (\mathbf{0}, \mathbf{0}, 1, 1)^T$$

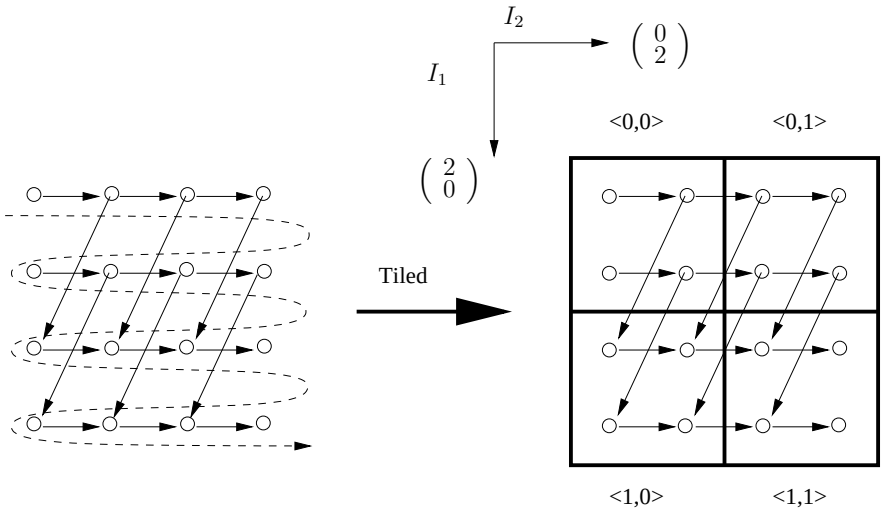


(a) An original space $\mathcal{I}$

(b) A tiled space $\mathcal{I}_{tiled}$

Figure 6.2: Tiling with $B_2 = \left((3, 0)^T, (2, 0)^T\right)$.

In Figure 6.2, the dependence matrix D is $\begin{pmatrix} 0 & 2 \\ 1 & -1 \end{pmatrix}$, and tiling space matrix B_2 is $\begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$. An iteration point $\vec{c_1}$, which is specified by $(0,2)^T$ in an original space, belongs to the tile $< 0,1 >$. All iteration points in the tile $< 0,1 >$ will be executed after all iteration points in the tile $< 0,0 >$ are executed. However, in this tiling scheme it is not possible to respect all dependence relations. For example, an iteration point $\vec{c_2}$ in the tile $< 0,0 >$ depends on the iteration point $\vec{c_1}$ that belongs to the tile $< 0,1 >$ which is supposed to be executed after the tile $< 0,0 >$. Therefore, the dependence relation between iteration points $\vec{c_1}$ and $\vec{c_2}$ can not be respected in the tiled space. This violation of the dependence prohibits $B_2 = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$ from being used as the tiling matrix.



(a) An original space $\mathcal{I}$

(b) A tiled space $\mathcal{I}_{tiled}$

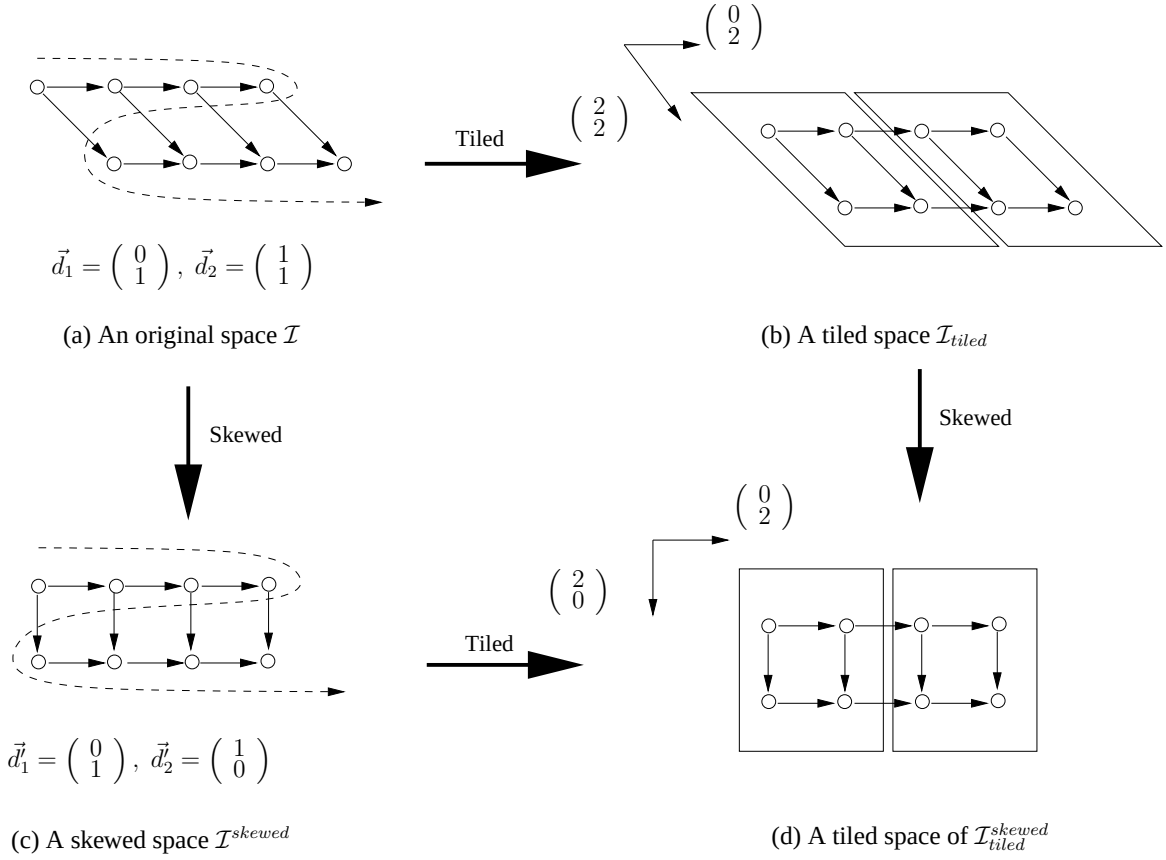
Figure 6.3: Tiling with $B_1 = \left((2,0)^T, (2,0)^T\right)$.

In Figure 6.3, a different tiling scheme is applied, in which the tiling matrix $B_1 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ is used. All dependence relations are respected by following the execution order of the tiled space. When B is the tiling matrix, and $\vec{c}$ is an iteration point in an original space, $\lfloor B^{-1}\vec{c} \rfloor$ gives the tile to which $\vec{c}$ belongs in the tiled space. For the rest of this chapter, we write B^{-1} as the matrix U. The problem of violating a dependence relation in Figure 6.2 can be clearly explained by finding tiles of iteration points $\vec{c_1}$ and $\vec{c_2}$. The tile to which $\vec{c}_1$ is mapped should lexicographically precede the tile to which $\vec{c}_2$ is mapped.

$$
\begin{aligned}
\lfloor U\vec{c_1} \rfloor &= \left\lfloor \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right\rfloor \\
&= \left\lfloor \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\rfloor \\
&= \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \\
\lfloor U\vec{c_2} \rfloor &= \left\lfloor \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \right\rfloor \\
&= \left\lfloor \begin{pmatrix} \frac{2}{3} \\ \frac{1}{2} \end{pmatrix} \right\rfloor \\
&= \begin{pmatrix} 0 \\ 0 \end{pmatrix}.
\end{aligned}
$$

The tile $< 0, 0 >$ for $\vec{c}_2$ lexicographically precedes the tile $< 0, 1 >$ for $\vec{c}_1$. So, the tiling matrix $B_2 = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$ can not respect the dependence $(\vec{c_2} - \vec{c_1})$.

Loop skewing is one of the common compiler transformation techniques. Skewing changes the shape of an iteration space. As long as the dependence vectors of the skewed iteration space are legal, skewing is legal. Figure 6.4 shows an original iteration space $\mathcal{I}$ and the skewed iteration space $\mathcal{I}^{skewed}$. As the dotted arrows show, the execution orders of iteration points in both iteration space $\mathcal{I}$, and $\mathcal{I}^{skewed}$ are exactly same. The dependence



(a) An original space $\mathcal{I}$

(b) A tiled space $\mathcal{I}_{tiled}$

(c) A skewed space $\mathcal{I}^{skewed}$

(d) A tiled space of $\mathcal{I}^{skewed}_{tiled}$

Figure 6.4: Skewing.

vectors of $\mathcal{I}$ are $\vec{d_1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\vec{d_2} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. The skewed space $\mathcal{I}^{skewed}$ has dependence vectors $\vec{d_1'} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\vec{d_2'} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, which are legal. Figure 6.4-(b) and (d) show the tiled spaces of $\mathcal{I}$ and $\mathcal{I}^{skewed}$. The tiled space $\mathcal{I}_{tiled}$ of an original iteration space $\mathcal{I}$ in Figure 6.4-(b) is specified by $\begin{pmatrix} 2 & 0 \\ 2 & 2 \end{pmatrix}$. The tiled space $\mathcal{I}_{tiled}^{skewed}$ of skewed iteration space $\mathcal{I}^{skewed}$ is specified by $\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$.

Definition 6.1 *When the tiling space matrix B is of the form $B = (\vec{b_1}\vec{b_2}\cdots\vec{b_n}), \vec{s_i} = b_i\vec{e}_i, b_i \geq 1, b_i \in I, \vec{e}_i$ is ith column of an identity matrix $I_{n\times n}$, B is called a normal form tiling matrix.*

When B is in the normal form,

$$\begin{aligned} B &= (\vec{b_1}\vec{b_2}\cdots\vec{b_n}) \\ &= \begin{pmatrix} b_1 & 0 & & 0 \\ 0 & b_2 & \cdots & \vdots \\ \vdots & \vdots & & 0 \\ 0 & 0 & & b_n \end{pmatrix}. \end{aligned}$$

Then,

$$U = \begin{pmatrix} \frac{1}{b_1} & 0 & & 0 \\ 0 & \frac{1}{b_2} & \cdots & \vdots \\ \vdots & \vdots & & 0 \\ 0 & 0 & & \frac{1}{b_n} \end{pmatrix}.$$

Non-rectangular tiling can be converted into rectangular tiling by applying skewing an iteration space $\mathcal{I}$ and then choosing a normal form tiling space matrix.

6.1 Dependences in Tiled Space

Proposition 6.1 *When $\vec{d} \xrightarrow{tiled} \left\{ \begin{pmatrix} \vec{t_1} \\ \vec{l_1} \end{pmatrix}, \cdots, \begin{pmatrix} \vec{t_r} \\ \vec{l_r} \end{pmatrix} \right\}$, $\vec{d} = B\vec{t_i} + \vec{l_i}$, $1 \leq i \leq r$, where r is a function of B and $\vec{d}$.*

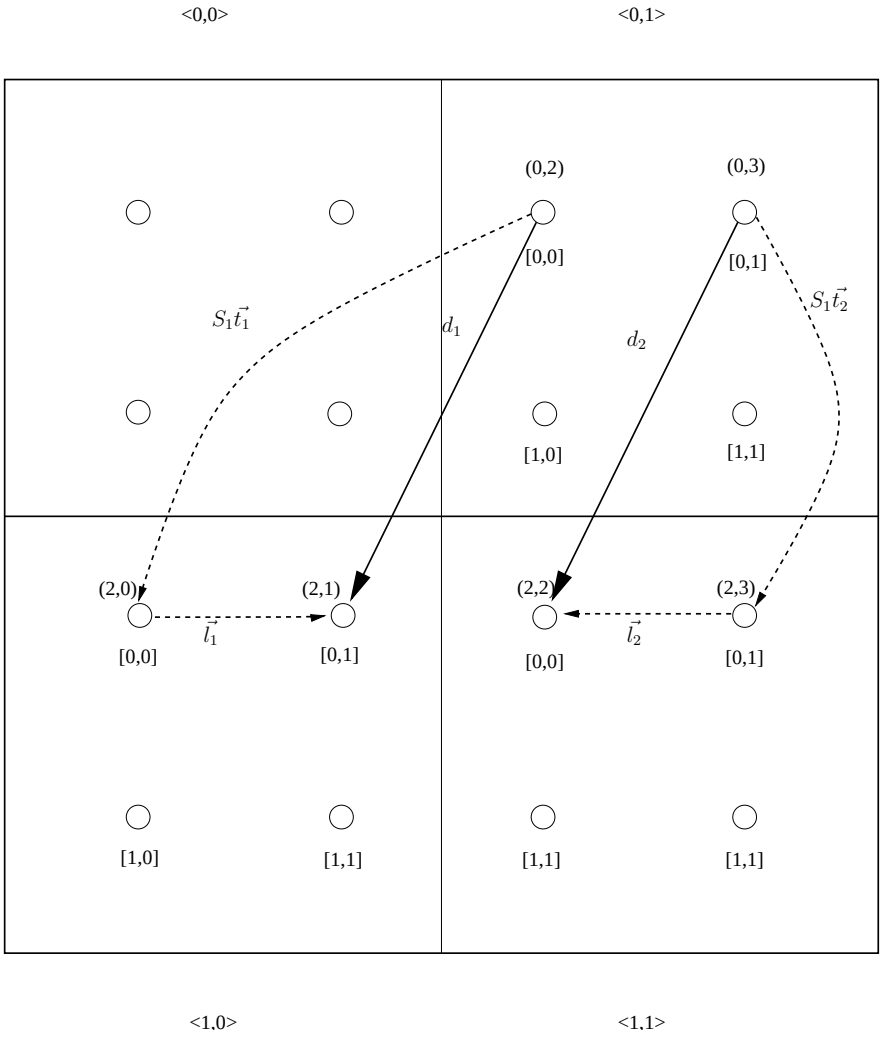


Figure 6.5: Illustration of $\vec{d} = B\vec{t} + \vec{l}$.

Figure 6.5 shows an example for Proposition 6.1. For simplicity, only two dependence vectors are captured. However, every iteration point except boundary points has same dependence patterns. Actually, $\vec{d_1}$ and $\vec{d_2}$ are same dependence vector, but their positions in an iteration space are different. $\vec{d_1}$ is defined between two iteration points $(0,2)^T$ and $(2,1)^T$, and $\vec{d_2}$ between $(0,3)^T$ and $(2,2)^T$. In this tiling scheme, the tiling space matrix $B = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ is used. An iteration point $(0,2)^T$ is mapped to $(0,1,0,0)^T$ in the tiled space, $(2,1)^T$ to $(1,0,0,1)^T$, $(0,3)^T$ to $(0,1,0,1)^T$, and $(2,2)^T$ to $(1,1,0,0)^T$. In the tiled space, the dependence vector is defined in the same way as in an original iteration space. Let $\mathcal{I}_{tiled}(sink(\vec{d_i}))$ and $\mathcal{I}_{tiled}(source(\vec{d_i}))$ be corresponding iteration points in the tiled space of the sink and the source of $\vec{d_i}$ in an original iteration space respectively.

$$\vec{d^i_{tiled}} = \mathcal{I}_{tiled}(sink(\vec{d_i})) - \mathcal{I}_{tiled}(source(\vec{d_i})).$$

The corresponding dependence vector, $\vec{d^1_{tiled}}$, in the tiled space is defined as follows.

$$\begin{aligned}
\vec{d_i} \xrightarrow{tiled} \vec{d^i_{tiled}} &= \begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix} \\
\vec{d^1_{tiled}} &= \mathcal{I}_{tiled}(sink(\vec{d_1})) - \mathcal{I}_{tiled}(source(\vec{d_1})) \\
&= \mathcal{I}_{tiled}((2,1)^T) - \mathcal{I}_{tiled}((0,2)^T) \\
&= \begin{pmatrix} \mathbf{1} \\ \mathbf{0} \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} \mathbf{0} \\ \mathbf{1} \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \mathbf{1} \\ \mathbf{-1} \\ 0 \\ 1 \end{pmatrix} \\
\vec{t_1} &= \begin{pmatrix} \mathbf{1} \\ \mathbf{-1} \end{pmatrix} \\
\vec{l_1} &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
\vec{d^2_{tiled}} &= \mathcal{I}_{tiled}(sink(\vec{d_2})) - \mathcal{I}_{tiled}(source(\vec{d_2}))
\end{aligned}$$

$$
\begin{aligned}
& = \mathcal{I}_{tiled}((2,2)^T) - \mathcal{I}_{tiled}((0,3)^T) \\
\vec{d}^2_{tiled} & = \begin{pmatrix} \mathbf{1} \\ \mathbf{1} \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} \mathbf{0} \\ \mathbf{1} \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{1} \\ \mathbf{0} \\ 0 \\ -1 \end{pmatrix} \\
\vec{t}_2 & = \begin{pmatrix} \mathbf{1} \\ \mathbf{0} \end{pmatrix} \\
\vec{l}_2 & = \begin{pmatrix} 0 \\ -1 \end{pmatrix}.
\end{aligned}
$$

Proposition 6.1 shows that the relation between a dependence vector in $\mathcal{I}$ and its corresponding dependence vector in $\mathcal{I}_{tiled}$. Figure 6.5 shows that an iteration point $(0,2)^T$, the source of $\vec{d}_1$, in an original space is mapped to an iteration point $(2,0)^T$ in an original space by $B_1\vec{t}_1$.

$$
\begin{aligned}
& B_1\vec{t}_1 + \text{ the source of } \vec{d}_1 \\
= & \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \end{pmatrix} \\
= & \begin{pmatrix} 2 \\ 0 \end{pmatrix}.
\end{aligned}
$$

By adding $B\vec{t}$ to an iteration point $\vec{c}_\alpha$, $\vec{c}_\alpha$ is mapped to the iteration point $\vec{c}_\beta$ in the different tile, when $\vec{t} \neq \vec{0}$ [2]. The intra-tile positions of $\vec{c}_\alpha$ and of $\vec{c}_\beta$ within their own tiles are same. For example, an iteration point $(0,2)^T$ is located in $[0,0]$ of the tile $< 0,1 >$, and an iteration point $(2,0)^T$ is located in $[0,0]$ of the tile $< 1,0 >$. By adding intra-tile vector $\vec{l}_1$ to the iteration point $(2,0)^T$, an iteration point $(2,1)^T$, the sink of $\vec{d}_1$, in an original space

[2] In case of $\vec{t} = \vec{0}$, $\vec{d}_{tiled} = \begin{pmatrix} \vec{0} \\ \vec{l} \end{pmatrix}$, which is a trivial case.

is reached.

$$
\begin{aligned}
& B_1\vec{t}_1 + \text{ the source of } \vec{d}_1 + \vec{l}_1 \\
= & \begin{pmatrix} 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
= & \begin{pmatrix} 2 \\ 1 \end{pmatrix} \\
= & \text{ the sink of } \vec{d}_1 \\
\Rightarrow B_1\vec{t}_1 + \vec{l}_1 = & \text{ the sink of } \vec{d}_1 - \text{ the source of } \vec{d}_1 \\
= & \begin{pmatrix} 2 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \\
= & \begin{pmatrix} 2 \\ -1 \end{pmatrix} \\
= & \vec{d}_1.
\end{aligned}
$$

Similarly, $\vec{d}_2$ can be expressed as follows.

$$
\begin{aligned}
\vec{d}_2 &= B_1\vec{t}_2 + \vec{l}_2 \\
\begin{pmatrix} 2 \\ -1 \end{pmatrix} &= \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}.
\end{aligned}
$$

6.2 Legality of Tiling

Definition 6.2 *Let $\vec{x} = (x_1, x_2, \cdots, x_n)^T$ and $\vec{y} = (y_1, y_2, \cdots, y_n)^T$ be n-dimension vectors. When there is i, $1 \leq i \leq n-1$ such that $x_j = y_j$ and $x_i < y_i$ for $j,\ 1 \leq j \leq i-1$, it is said that $\vec{x}$ lexicographically precedes $\vec{y}$, which is denoted with $\vec{x} \prec_{lex} \vec{y}$.*

Definition 6.3 *When $\vec{0} \prec_{lex} \vec{x}$, it is said that a vector $\vec{x}$ is lexicographically positive.*

Definition 6.4 *When* $\vec{i} = (i_1, i_2, \cdots, i_n)^T, i_j \in R, 1 \leq j \leq n$, $\lfloor \vec{i} \rfloor$ *means applying* $\lfloor \rfloor$ *to every element in* $\vec{i}$. $\lfloor \vec{i} \rfloor = (\lfloor i_1 \rfloor, \lfloor i_2 \rfloor, \cdots, \lfloor i_n \rfloor)^T$. *By the definition of* $\lfloor \rfloor$, *we may define* $\lfloor \vec{i} \rfloor$ *by applying* $\lfloor \rfloor$ *to every element except integer elements in* $\vec{i}$.

Theorem 6.1 *Tiling is legal if and only if* $\vec{t_i}$*'s are legal or* $\vec{t_i} = \vec{0}$.

(Proof) ($\Rightarrow$) If tiling is legal, all dependence vectors $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix}$, $1 \leq i \leq r$ in the tiled space are legal. If $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix}$ is legal, then $\vec{t_i}$ is legal or ($\vec{t_i} = \vec{0}$ and $\vec{d} = \vec{l_i}$). When ($\vec{t_i} = \vec{0}$ and $\vec{d} = \vec{l_i}$), $\vec{l_i}$ is legal by the definition of $\vec{d}$.
($\Leftarrow$) When $\vec{t_i}$ is legal for $1 \leq i \leq r$, $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix}$ is legal for all i, $1 \leq i \leq r$. When $\vec{t_i} = \vec{0}$, the dependence vector in the tiled space is $\begin{pmatrix} \vec{0} \\ \vec{l_i} \end{pmatrix} = \begin{pmatrix} \vec{0} \\ \vec{d} \end{pmatrix}$. By the definition of $\vec{d}$, $\vec{l_i}$ is legal. So, $\begin{pmatrix} \vec{0} \\ \vec{l_i} \end{pmatrix}$ is legal. Therefore, tiling is legal.

Lemma 6.1 *If each* $\vec{t_i}$ *from* $\left\{ \begin{pmatrix} \vec{t_1} \\ \vec{l_1} \end{pmatrix}, \cdots, \begin{pmatrix} \vec{t_r} \\ \vec{l_r} \end{pmatrix} \right\}, \vec{t_i}, \vec{l_i} \in I^n$, *is nonnegative* ($\vec{t_i} \geq \vec{0}$), *then either* $\lfloor U\vec{d} \rfloor = \lfloor B^{-1}\vec{d} \rfloor$ *is positive, or* $\vec{t_i} = \vec{0}$ *and* $\vec{d} = \vec{l_i}$.

(Proof) There are the following two cases:

$$\vec{t_i} > \vec{0}$$

$$\vec{t_i} = \vec{0}.$$

Let $\vec{x} = (x_1, x_2, \cdots, x_n)^T = U\vec{d}$ and $\vec{y_i} = (y_{i,1}, y_{i,2}, \cdots, y_{i,n})^T = Ul_i, 1 \leq i \leq r$. $\vec{x}$ and $\vec{y_i}$ is real vectors ($\vec{x}, \vec{y_i} \in R^n$), but from the Proposition 6.1, $\vec{t_i} = (\vec{x} - \vec{y_i})$ is an integer vector($(\vec{x} - \vec{y_i}) \in I^n$).

In the first case,

$$\begin{aligned}
\vec{t_i} &= U\vec{d} - U\vec{l_i} > \vec{0},\ (U\vec{d} - U\vec{l_i}) \in I^n \\
\vec{t_i} &= \lfloor \vec{t_i} \rfloor \text{ (Since } \vec{t_i} \text{ is integral)} \\
\vec{t_i} &= \lfloor U\vec{d} - U\vec{l_i} \rfloor \geq \vec{1} \\
&= \lfloor \vec{x} - \vec{y_i} \rfloor \geq \vec{1} \\
&= \begin{pmatrix} \lfloor x_1 - y_{i,1} \rfloor \\ \vdots \\ \lfloor x_n - y_{i,n} \rfloor \end{pmatrix} \geq \vec{1}, (|y_{i,j}| < 1,\ 1 \leq i \leq r, 1 \leq j \leq n).
\end{aligned}$$

Then

$$\begin{aligned}
\lfloor x_j - y_{i,j} \rfloor &\geq 1, |y_{i,j}| < 1, (1 \leq i \leq r, 1 \leq j \leq n) \\
x_j - y_{i,j} &\geq 1,\ (-1 < y_{i,j} < 1) \\
x_j &\geq y_{i,j} + 1, (0 < y_{i,j} + 1 < 2).
\end{aligned}$$

So,

$$\begin{aligned}
x_j > 0, (1 \leq j \leq n) &\Rightarrow \vec{x} > \vec{0} \\
&\Rightarrow U\vec{d} > \vec{0} \\
&\Rightarrow \lfloor U\vec{d} \rfloor \geq \vec{0}.
\end{aligned}$$

In the second case,

$$\vec{t_i} = U\vec{d} - U\vec{l_i} = \vec{0}, 1 \leq i \leq r$$

$$
\begin{aligned}
U\vec{d} &= U\vec{l_i} \\
\vec{d} &= \vec{l_i},
\end{aligned}
$$

So, if $\vec{t_i}, 1 \leq i \leq r$ is lexicographically positive($\geq \vec{0}$), then $\lfloor U\vec{d} \rfloor \geq \vec{0}$ or $\vec{d} = \vec{l_i}$.

Lemma 6.2 *$\lfloor U\vec{d} \rfloor \geq \vec{0}$ is a sufficient condition for* $\left\{ \begin{pmatrix} \vec{t_1} \\ \vec{l_1} \end{pmatrix}, \cdots, \begin{pmatrix} \vec{t_r} \\ \vec{l_r} \end{pmatrix} \right\}$ *to be all legal dependence vectors.*

(Proof) We know that $\vec{d} = B\vec{t_i} + \vec{l_i}$, $1 \leq i \leq r$. Let $\vec{y_i} = (y_{i,1}, y_{i,2}, \cdots, y_{i,n})^T = U\vec{l_i}$. $\vec{t_i} = (t_{i,1}, t_{i,2}, \cdots, t_{i,n})^T$ is an integer vector.

If $\lfloor U\vec{d} \rfloor \geq \vec{0}$, then

$$
\begin{aligned}
U\vec{d} &= \vec{t_i} + U\vec{l_i} \\
\lfloor U\vec{d} \rfloor &= \lfloor \vec{t_i} + U\vec{l_i} \rfloor \geq \vec{0} \\
&= \begin{pmatrix} \lfloor t_{i,1} + y_{i,1} \rfloor \\ \vdots \\ \lfloor t_{i,n} + y_{i,n} \rfloor \end{pmatrix} \geq \vec{0}.
\end{aligned}
$$

Then,

$$
\begin{aligned}
\lfloor t_{i,j} + y_{i,j} \rfloor &\geq 0, \ (1 \leq i \leq r, \ 1 \leq j \leq n) \\
t_{i,j} + y_{i,j} &\geq 0, \ (|y_{i,j}| < 1) \\
t_{i,j} &\geq -y_{i,j}, \ (-1 < -y_{i,j} < 1) \\
t_{i,j} &> -1.
\end{aligned}
$$

$t_{i,j}$ is an integer such that $t_{i,j} > -1$. So, $t_{i,j} \geq 0 \wedge t_{i,j} \in I$. It means $\vec{t_i} \geq \vec{0}$. When $\vec{t_i} > \vec{0}$, $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix}$ is a legal dependence vector. When $\vec{t_i} = \vec{0}$, $\vec{d} = \vec{l_i}$. So, $\begin{pmatrix} \vec{0} \\ \vec{l_i} \end{pmatrix} = \begin{pmatrix} \vec{0} \\ \vec{d} \end{pmatrix}$ is also legal because $\vec{d}$ is legal. Therefore, $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix}$ is a legal dependence vector, if $\lfloor U\vec{d} \rfloor \geq \vec{0}$.

The legality of dependence vector allows for $\vec{d}$ to have negative elements. The legality of $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix}$ does not necessarily means $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix} \geq \vec{0}$. So, $\lfloor U\vec{d} \rfloor \geq \vec{0}$ does not mean $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix} \geq \vec{0}$, even if it guarantees the legality of $\begin{pmatrix} \vec{t_i} \\ \vec{l_i} \end{pmatrix}$.

Theorem 6.2 *$\lfloor U\vec{d} \rfloor \geq \vec{0}$ is a necessary and sufficient condition for $\vec{t_i} \geq \vec{0}$*

(Proof) It's clear from Lemma 6.1 and Lemma 6.2.

Corollary 6.1 *When the tiling space matrix B is of the normal form, if $\vec{d} \geq \vec{0}$, then tiling with B is legal.*

(Proof) When B is a normal form matrix, and $\vec{d} \geq \vec{0}$, it is guaranteed that $\lfloor U\vec{d} \rfloor \geq \vec{0}$. From Theorem 6.2, tiling is legal.

Theorem 6.3 *For any real numbers a and b, $\lfloor a - b \rfloor \leq \lfloor a \rfloor - \lfloor b \rfloor$.*

(Proof) For any real number x, we define fracpart(x) as $x - \lfloor x \rfloor$. By definition, $0 \leq$ fracpart$(x) < 1$. Thus,

$$\begin{aligned} \lfloor a - b \rfloor &= \lfloor \lfloor a \rfloor + \text{fracpart}(a) - \lfloor b \rfloor - \text{fracpart}(b) \rfloor \\ &= \lfloor a \rfloor - \lfloor b \rfloor + \lfloor \text{fracpart}(a) - \text{fracpart}(b) \rfloor \end{aligned}$$

Since $0 \leq$ fracpart$(a) < 1$, and $0 \leq$ fracpart$(b) < 1$, it follows that $-1 <$ fracpart$(a) -$ fracpart$(b) < 1$. Therefore, $\lfloor$fracpart$(a) -$ fracpart$(b)\rfloor$ is either 0 or -1. Hence, the result. Also note that $\lfloor a - b\rfloor = \lfloor a\rfloor - \lfloor b\rfloor$ if and only if fracpart$(a) =$ fracpart(b).

When there is a dependence relation between two iteration points $\vec{c}_1$ and $\vec{c}_2$, the dependence relation can be expressed by $\vec{c}_2 = \vec{c}_1 + \vec{d}$. Even in the tiled space, the dependence relation in the original iteration space should be respected. Otherwise, the tiling is illegal. The tile to which an iteration point $\vec{c}_1$ is mapped should be executed before the tile to which $\vec{c}_2$ is mapped. The difference vector $\vec{t}$ between these two tiles can be expressed as follows.

$$\begin{aligned} \vec{t} &= \lfloor U\vec{c}_2\rfloor - \lfloor U\vec{c}_1\rfloor \\ &= \lfloor U(\vec{c}_1 + \vec{d})\rfloor - \lfloor U\vec{c}_1\rfloor \qquad (6.1) \\ &\geq \lfloor U(\vec{c}_1 + \vec{d}) - U\vec{c}_1\rfloor \text{ (By Theorem 6.3)} \\ &= \lfloor U\vec{d}\rfloor \\ \Rightarrow \vec{t} &\geq \lfloor U\vec{d}\rfloor \qquad (6.2) \end{aligned}$$

We can find all possible tile vector $\vec{t}$ by applying Equation 6.1 to all iteration points that belong to the same tile. For example, Figure 6.6 shows a part of Figure 6.2. The tile $< 0, 1 >$ contains 6 iteration points, $\{(0,2)^T, (0,3^T), (1,2)^T, (1,3)^T, (2,2)^T, (2,3)^T\}$. Because all iteration points except boundary points have the same dependence pattern, we can find all possible tile vectors, $\vec{t}$ by taking care of all iteration points in a single specific tile. Let $\mathcal{T}_{\vec{d}}$ be the set of all possible $\vec{t}$ for a dependence vector $\vec{d}$.

$$\mathcal{T}_{\vec{d}} = \{\vec{t} | \vec{t} = \lfloor U(\vec{i} + \vec{d})\rfloor - \lfloor U\vec{i}\rfloor,\ \forall \vec{i} \in \text{a specific tile}\}.$$

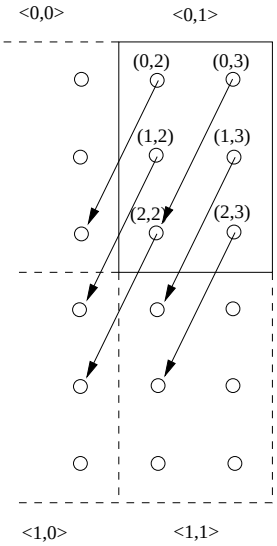


Figure 6.6: An example for $\mathcal{T}_{\vec{d}}$.

For the tiling scheme in Figure 6.6,

$$\mathcal{T}_{\vec{d}} = \left\{ \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}.$$

Let $(U\vec{d})[k]$ be the kth element of $U\vec{d}$. When $|(U\vec{d})[k]| < 1, 1 \leq k \leq n$, $\lfloor (U\vec{d})[k] \rfloor$ is either 0 or -1, and $\lceil (U\vec{d})[k] \rceil$ is either 0 or 1. $\mathcal{T}_{\vec{d}}$ can be found by applying all possible combinations of $\lfloor \rfloor$ and $\lceil \rceil$ to the elements of $U\vec{d}$. When α is an integer, $\lfloor \alpha \rfloor = \lceil \alpha \rceil = \alpha$. Therefore, we need to take care of non-integral elements in $U\vec{d}$. So, the size of $\mathcal{T}_{\vec{d}}$ is 2^r, where r is the number of non-integral elements in $U\vec{d}$. In Figure 6.6, $U_2 = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$, and $\vec{d} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$.

$$U_2\vec{d} = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{2}{3} \\ \frac{-1}{2} \end{pmatrix}.$$

So,

$$\begin{aligned} \mathcal{T}_{\vec{d}} &= \left\{ \begin{pmatrix} \lfloor \frac{2}{3} \rfloor \\ \lfloor \frac{-1}{2} \rfloor \end{pmatrix}, \begin{pmatrix} \lfloor \frac{2}{3} \rfloor \\ \lceil \frac{-1}{2} \rceil \end{pmatrix}, \begin{pmatrix} \lceil \frac{2}{3} \rceil \\ \lfloor \frac{-1}{2} \rfloor \end{pmatrix}, \begin{pmatrix} \lceil \frac{2}{3} \rceil \\ \lceil \frac{-1}{2} \rceil \end{pmatrix} \right\} \\ &= \left\{ \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}. \end{aligned}$$

Definition 6.5 *When the first non-zero element of a vector $\vec{i}$ is non-negative, the vector $\vec{i}$ is called a legal vector, or the vector $\vec{i}$ is legal.*

Definition 6.6 *When the dependence vector $\vec{d}$ in the original iteration space is preserved in the tiled space, it is said that tiling is legal for the dependence vector $\vec{d}$.*

Lemma 6.3 *If $\lfloor U\vec{d} \rfloor$ is legal, then tiling is legal for a dependence vector $\vec{d}$.*

(Proof) When $\mathcal{T}_{\vec{d}}$ contains only legal vectors, tiling is legal. From the Equation 6.2, we know that $\lfloor U\vec{d} \rfloor$ belongs to $\mathcal{T}_{\vec{d}}$ and that $\lfloor U\vec{d} \rfloor$ is the earliest vector lexicographically in $\mathcal{T}_{\vec{d}}$, which means that other tile vectors $\vec{t}$ are legal, if $\lfloor U\vec{d} \rfloor$ is legal. So, $\mathcal{T}_{\vec{d}}$ contains only legal vectors. Therefore, if $\lfloor U\vec{d} \rfloor$ is legal, then tiling is legal.

Lemma 6.4 *For an iteration space with the dependence matrix $D = (\vec{d_1}, \vec{d_2}, \cdots, \vec{d_p})$, if $\lfloor U\vec{d_i} \rfloor$ is legal for all $i,\ 1 \leq i \leq p$, then tiling is legal.*

(Proof) It is clear from Lemma 6.3.

Lemma 6.5 *When $-1 < (U\vec{d})[k] < 1, 1 \leq k \leq n$, if $(U\vec{d})[k]$ is negative for some k, then tiling is illegal for a dependence vector $\vec{d}$.*

(Proof) $\mathcal{T}_{\vec{d}}$ always contains $\lfloor U\vec{d} \rfloor$ as its member. $\mathcal{T}_{\vec{d}}$ should contain only legal tile vectors in order for tiling is legal. When $-1 < (U\vec{d})[k] < 1, 1 \leq k \leq n$, the only possible value that $\lfloor (U\vec{d})[k] \rfloor$ can have is either 0 or -1. $\lfloor U\vec{d} \rfloor$ consists of only 0 and -1. So, when $(U\vec{d})[k]$ is negative, $\lfloor U\vec{d} \rfloor$ contains at least one -1 for kth element. Then, it is guaranteed that at least one vector in $\mathcal{T}_{\vec{d}}$ is illegal. Therefore, tiling is illegal.

Theorem 6.4 *When $-1 < (U\vec{d})[k] < 1, 1 \leq k \leq n$, the nonnegativity of every element of $U\vec{d}$ is a necessary and sufficient condition for tiling for a dependence vector $\vec{d}$.*

(Proof) It is clear from the Lemma 6.3 and Lemma 6.5.

Corollary 6.2 *For an iteration space with the dependence matrix $D = (\vec{d_1}, \vec{d_2}, \cdots, \vec{d_p})$, when $-1 < (U\vec{d_i})[k] < 1, 1 \leq i \leq p, 1 \leq k \leq n$, the nonnegativity of every element of $U\vec{d_i}, 1 \leq i \leq p$ is a necessary and sufficient condition for tiling.*

(Proof) It is clear from the Theorem 6.4.

When $D = (\vec{d_1}, \vec{d_2}, \cdots, \vec{d_p}),\ p \geq 2$ is a dependence matrix in two dimensional iteration space, each dependence vector $\vec{d_i}$ can be specified by nonnegative linear combination of two extreme vectors. Let $\vec{r_1} = \begin{pmatrix} r_{11} \\ r_{21} \end{pmatrix}$ and $\vec{r_2} = \begin{pmatrix} r_{12} \\ r_{22} \end{pmatrix}$ be two extreme vectors from D. Then,

$$\vec{d_i} = \alpha\vec{r_1} + \beta\vec{r_2},\ (\alpha \geq 0, \beta \geq 0, \alpha, \beta \in R, 1 \leq i \leq p).$$

Theorem 6.5 *Tiling with $B = (\vec{r_1}\vec{r_2})$ in two dimensional iteration space is legal.*

(Proof) $B = \begin{pmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{pmatrix}$; $\vec{d_i} = \begin{pmatrix} \alpha_i r_{11} + \beta_i r_{12} \\ \alpha_i r_{21} + \beta_i r_{22} \end{pmatrix}$.
$U = B^{-1} = \frac{1}{\Delta} \begin{pmatrix} r_{22} & -r_{12} \\ -r_{21} & r_{11} \end{pmatrix}$, where $\Delta = r_{11}r_{22} - r_{12}r_{21}$.

$$U\vec{d_i} = \frac{1}{\Delta} \begin{pmatrix} r_{22} & -r_{12} \\ -r_{21} & r_{11} \end{pmatrix} \begin{pmatrix} \alpha_i r_{11} + \beta_i r_{12} \\ \alpha_i r_{21} + \beta_i r_{22} \end{pmatrix},\ (1 \leq i \leq p)$$

$$
\begin{aligned}
&= \frac{1}{\Delta}\begin{pmatrix} \alpha_i r_{11}r_{22} + \beta_i r_{12}r_{22} - \alpha_i r_{12}r_{21} - \beta_i r_{12}r_{22} \\ -\alpha_i r_{11}r_{21} - \beta_i r_{12}r_{21} + \alpha_i r_{11}r_{21} + \beta_i r_{11}r_{22} \end{pmatrix} \\
&= \frac{1}{\Delta}\begin{pmatrix} \alpha_i(r_{11}r_{22} - r_{12}r_{21}) \\ \beta_i(r_{11}r_{22} - r_{12}r_{21}) \end{pmatrix} \\
&= \begin{pmatrix} \alpha_i \\ \beta_i \end{pmatrix} \geq \vec{0} \\
\Rightarrow \lfloor U\vec{d_i} \rfloor &= \begin{pmatrix} \lfloor \alpha_i \rfloor \\ \lfloor \beta_i \rfloor \end{pmatrix} \\
&\geq \vec{0}
\end{aligned}
$$

From Lemma 6.2, tiling is legal.

It is easy to know that $B = (\vec{r_1}\vec{r_2})$ may not be in a normal form.

6.3 An Algorithm for Tiling Space Matrix

From Corollary 6.1, we just need to take care of dependence vectors that have negative element(s) in order to find a normal form tiling space matrix.

[Example] Let $D = \begin{pmatrix} 1 & 1 & 2 \\ 3 & 1 & -1 \\ -2 & 2 & 3 \end{pmatrix}$. We need to take care of dependence vectors that have negative element(s). $D' = \begin{pmatrix} 1 & 2 \\ 3 & -1 \\ -2 & 3 \end{pmatrix}$. D' is arranged by the level of first negative element. $D'' = \begin{pmatrix} 2 & 1 \\ -1 & 3 \\ 3 & -2 \end{pmatrix}$. At the first iteration of $while$ loop, $\vec{d} = \begin{pmatrix} 2 \\ -1 \\ 3 \end{pmatrix}$. Here, $level(\vec{d})$ is 2. k is 2. The smallest integer value for α should be chosen such that $\lfloor \frac{d_{(k-1)}}{\alpha} \rfloor = \lfloor \frac{d_1}{\alpha} \rfloor = \lfloor \frac{2}{\alpha} \rfloor > 0$ and $\alpha > 1$. α is 2. $b_{(k-1)} = b_1$ is assigned 2. In a similar way at the second iteration, $\vec{d} = \begin{pmatrix} 1 \\ 3 \\ -2 \end{pmatrix}$, and k is 3. $\lfloor \frac{d_2}{\alpha} \rfloor = \lfloor \frac{3}{\alpha} \rfloor > 0$ and $\alpha > 1$. α is 3. So, $b_{(s-1)} = b_2 = 3$. All columns in D'' are processed. A normal form tiling matrix

Procedure Find_Tiling(D)
D : A dependence matrix
begin
  $D' \leftarrow$ dependence vectors with negative element;
  $D'' \leftarrow$ Arrange column vector in D' by the level of first negative element;
  Initialize B by assigning 0 to all elements of B;

  while (D'' is non-empty)
    $\vec{d} \leftarrow$ first column vector in D'';
    $D'' \leftarrow D'' - \{\vec{d}\}$;
    $k \leftarrow$ level of first negative element of $\vec{d}$;

    if ($d_{(k-1)} = 1$) **then**
      $\alpha \leftarrow 1$;
    else
      Find the smallest integer number α such that $\lfloor \frac{d_{(k-1)}}{\alpha} \rfloor > 0$ and $\alpha > 1$;
    endif

    if ($b_{(k-1)} > 0$) **then** /* $b_{(k-1)}$ is already assigned a value. */
      if ($b_{(k-1)} > \alpha$) **then** /* If several vectors have negative element */
        $b_{(k-1)} \leftarrow \alpha$; /* at the same level, the smallest α should be chosen. */
      endif
    else
      $b_{(k-1)} \leftarrow \alpha$;
    endif
  endwhile

  return B;
end

Figure 6.7: Algorithm for a normal form tiling space matrix B.

$B = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & b_3 \end{pmatrix}$ is found.

$$\begin{aligned} \lfloor UD \rfloor &= \left\lfloor \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{b_3} \end{pmatrix} \begin{pmatrix} 1 & 1 & 2 \\ 3 & 1 & -1 \\ -2 & 2 & 3 \end{pmatrix} \right\rfloor \\ &= \left\lfloor \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 1 \\ 1 & \frac{1}{3} & \frac{-1}{3} \\ \frac{-2}{b_3} & \frac{2}{b_3} & \frac{3}{b_3} \end{pmatrix} \right\rfloor \\ &= \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ \lfloor \frac{-2}{b_3} \rfloor & \lfloor \frac{2}{b_3} \rfloor & \lfloor \frac{3}{b_3} \rfloor \end{pmatrix} \end{aligned}$$

All column vectors in $\lfloor UD \rfloor$ are legal. So, from the Lemma 6.4, tiling with B is legal. The returned tiling matrix B may contain $b_i = 0$. In that case, B is not of normal form. If the returned B contain $b_i = 0$, we can assign any positive integer value to such b_i in order to make B be of normal form because those dimensions with $b_i = 0$ do not hurt legality of tiling.

6.4 Chapter Summary

We have found a sufficient condition and also a necessary and sufficient for tiling under a specific constraint. Based on the sufficient condition for tiling, we proposed an algorithm to find a legal tiling space matrix.

When a tiling space matrix B is of a normal form, the determinant of B is $|det(B)| = \Pi_{i=1}^{n} b_i$. Here, $|det(B)|$ is the size of a tile, the number of iteration space points that belong to a tile. Our algorithm considers only legality condition to find B. However, determining the size of a tile is a more complicated problem than it appears. When on-chip memory of embedded systems is not large enough to hold all necessary data, tiling should be considered

as an option to overcome the shortage of on-chip memory before an entire embedded system is re-designed. Obviously, tiling requires several accesses to off-chip memory, which will impose severe penalty on execution time as well as power consumption. To minimize the penalty caused by accesses to off-chip memory,it is needed to minimize the number of accesses to off-chip memory, which means that when we choose a tiling space matrix B, $|det(B)|$ should be as close as, but not larger than the size of on-chip memory. After B is founded by using our algorithm, if there is $b_i = 0$ in B, then ith dimension is a don't-care condition, because it does not hurt the legality of tiling. By adjusting the size of a tile in those don't-care dimensions, we can make the size of a tile as close as the size of on-chip memory. That adjustment will be considered in our future work.

Tiling is more compelling in general purpose systems than in embedded systems. In general purpose systems, the selection of tile sizes [18, 24, 45] is very closely related with some hardware features like the cache size and the cache line size and some interference misses like self-interference and cross-interference between data arrays [16, 31, 79]. Including those factors into our algorithm may help to find better tile size for general purpose systems.

CHAPTER 7

CONCLUSIONS

This thesis addresses several problems in the optimization of programs for embedded systems. The processor core in an embedded system plays an increasingly important role in addition to the memory sub-system. We focus on embedded digital signal processors (DSPs) in this work.

In Chapter 2, we have proposed and evaluated an algorithm to construct a worm partition graph by finding a longest worm at the moment and maintaining the legality of scheduling. Worm partitioning is very useful in code generation for embedded DSP processors. Previous work by Liao [51, 54] and Aho et al. [1] have presented expensive techniques for testing legality of schedules derived from worm partitioning. In addition, they do not present an approach to construct a legal worm partition of a DAG. Our approach is to guide the generation of legal worms while keeping the number of worms generated as small as possible. Our experimental results show that our algorithm can find most reduced worm partition graph as much as possible. By applying our algorithm to real problems, we find that it can effectively exploit the regularity of real world problems. We believe that this work has broader applicability in general scheduling problems for high-level synthesis.

Proper assignment of offsets to variables in embedded DSPs plays a key role in determining the execution time and amount of program memory needed. Chapter 3 proposes

a new approach of introducing a weight adjustment function and showed that its experimental results are slightly better and at least as well as the results of the previous works. More importantly, we have introduced a new way of handling the same edge weight in an access graph. As the SOA algorithm generates several fragmented paths, we show that the optimization of these path partitions is crucial to achieve an extra gain, which is clearly captured by our experimental results. We also have proposed usage of frequencies of variables in a GOA problem. Our experimental results show that this straightforward method is better than the previous research works.

In our weight adjustment functions, we handled Preference and Interference uniformly. We applied our weight adjustment functions to random data. Real-world algorithms, however, may have some patterns that are unique to each specific algorithm. We think that we may get a better result by introducing tuning factors an then handling Preference and Interference differently according to the pattern or the regularity in a specific algorithm. For example, when $(\alpha \cdot \text{Preference})/(\beta \cdot \text{Interference})$ is used as a weight adjustment function, setting $\alpha = \beta = 1$ gives our original weight adjustment functions. Finding optimal values of tuning factors may requires exhaustive simulation and take a lot of execution time for each algorithm.

In addition to offset assignment, address register allocation is important for embedded DSPs. In Chapter 4, we have developed an algorithm that can eliminate the explicit use of address register instructions in a loop. By introducing a compatible graph, our algorithm tries to find the most beneficial partitions at the moment. In addition, we developed an algorithm to find a lower bound on the number of ARs by finding the strong connected components (SCCs) of an extended graph. We implicitly assume that unlimited number of ARs are available in the AGU. However, usually it is not the case in real embedded systems

in which only limited number of ARs are available. Our algorithm tries to find partitions of array references in such a way that ARs cover as many array references as possible, which leads to minimization of the number of ARs needed. With the limited number of ARs, when the number of ARs needed to eliminate the explicit use of AR instructions is larger than the number of ARs available in the AGU, it is not possible to eliminate AR instructions in a loop. In that case, some partitions of array references should be merged in a way that the merger should minimize the number of explicit use of AR instructions. Our future works will be finding a model that can capture the effects of merging partitions on the explicit use of AR instructions. Based on that model, we will find efficient solution of AR allocation with the limited number of ARs.

When an array reference sequence becomes longer, and then the corresponding extended graph becomes denser, our lower bound on ARs with SCCs tended to be too optimistic. To prevent the lower bound from being too optimistic, we need to drop some back edges from the extended graph. In that case, it will be an important issue to determine which back edges should be dropped, which will be a focus of our future work.

Scheduling of computations and the associated memory requirement are closely interrelated for loop computations. Chapter 5 addresses this problem. In this chapter, we have developed a framework for studying the trade-off between scheduling and storage requirements. We developed methods to compute the region of feasible schedules for a given storage vector. In previous work, Strout et al. [74] have developed an algorithm for computing the universal occupancy vector which is the storage vector that is legal for any schedule of the iterations. By this, Strout et al. [74] mean any topological ordering of the nodes of an iteration space dependence graph (ISDG). Our work is applicable to wavefront schedules of nested loops. An important problem in this area is the extension of this work to imperfectly

nested loops, a sequence of loop nests and to whole programs. These problems represent significant opportunities for important work.

Tiling has long been used to improve the memory performance of loops on general-purpose computing systems. Previous characterization of tiling led to the development of sufficient conditions for the legality of tiling based only on the shape of tiles. While it was conjectured that the sufficient condition would also become necessary for “large enough” tiles, there had been no precise characterization of what is “large enough.” Chapter 6 develops a new framework for characterizing tiling by viewing tiles as points on a lattice. This also leads to the development of conditions under the legality condition for tiling is both necessary and sufficient.

BIBLIOGRAPHY

[1] A. Aho, S.C. Johnson, and J. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM,* 24(1):146-160, 1977.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers, Principles, Techniques and Tools. Addison Wesley, Boston 1988.

[3] F. E. Allen and J. Cocke. A Catalogue of Optimizing Transformations. Design and Optimization of Compilers. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[4] G. Araujo. Code Generation Algorithms for Digital Signal Processors. PhD thesis, Princeton Department of EE, June 1997.

[5] G. Araujo, S. Malik, and M. Lee. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *Proceedings of 33rd ACM/IEEE Design Automation Conference,* pages 591-596, June 1996.

[6] G. Araujo, A. Sudarsanam, and S. Malik. Instruction Set Design and Optimization for Address Computation in DSP Architectures. In *Proceedings of the 9th International Symposium on System Synthesis,* pages 31-37, November 1997.

[7] S. Atri, J. Ramanujam, and M. Kandemir. Improving offset assignment on embedded processors using transformations. In *Proc. High Performance Computing–HiPC 2000,* pp. 367–374, December 2000.

[8] Sunil Atri, J. Ramanujam, and M. Kandemir. Improving variable placement for embedded processors. In *Languages and Compilers for Parallel Computing,* (S. Midkiff et al. Eds.), Lecture Notes in Computer Science, vol. 2017, pp. 158–172, Springer-Verlag, 2001.

[9] D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys,* Vol. 26, No. 4, pages 345-420, December 1994.

[10] F. Balasa, F. Catthoor, and H.D. Man. Background memory area estimation for multidimensional signal processing systems. *IEEE Transactions on VLSI Systems,* 3(2):157-172, June 1995.

[11] U. Banerjee. Loop Parallelization. Kluwer Academic Publishers, 1994.

[12] D. Bartley. Optimization Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software Practice and Experience,* 22(2):101-110, February 1992.

[13] A. Basu, R. Leupers and P. Marwedel. Array Index Allocation under Register Constraints in DSP Programs. *12th Int. Conf. on VLSI Design,* GOA, India, Jan 1999.

[14] T. Ben Ismail, K. O'Brien, and A. Jerraya. Interactive System-level Partitioning with PARTIF. *Proc. of the European Design and Test Conference,* 1994.

[15] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal,* 17:33–51, 1994.

[16] Jacqueline Chame. Compiler Analysis of Cache Interference and its Applications to Compiler Optimizations. PhD thesis, Dept. of Computer Engineering, University of Southern California, 1997

[17] Y. Choi and T. Kim. Address assignment combined with scheduling in DSP code generation. in *Proc. 39th Design Automation Conference,* June 2002.

[18] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* pages 279-290, La Jolla, California, June 1995.

[19] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms,* MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1990.

[20] J. W. Davidson and C. W. Fraser. Eliminating Redundant Object Code. *In Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages,* pages 128-132, 1982.

[21] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[22] S. Devadas, A. Ghosh, and K. Keutzer. Logic Synthesis. McGraw Hill, New York, NY, 1994.

[23] J. Dongarra and R. Schreiber. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, May 1990.

[24] Karim Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[25] P. Feautrier. Array expansion. In *International Conference on Supercomputing,* pages 429-442, 1988.

[26] C. Fischer and R. LeBlanc. Crafting a Compiler with C. The Benjamin/Cummings Publishing Co., Redwood City, Ca, 1991.

[27] D. L. Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM,* 34(4):47-63, April 1991.

[28] D. Gajski, N. Dutt, S. Lin, and A. Wu. High Level Synthesis: Introduction to Chip and System Design. Kluwer Academic Publishers, 1992.

[29] J. G. Ganssle. The Art of Programming Embedded Systems. Academic Press, Inc., San Diego, California, 1992.

[30] DSP Address Optimization Using a Minimum Cost Circulation Technique. In In *Proceedings of International Conference on Computer-Aided Design,* pages 100–103, 1997.

[31] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 228-239, San Jose, California, October 1998

[32] G. Goossens, F. Catthoor, D. Lanneer, and H. De Man. Integration of Signal Processing Systems on Heterogeneous IC Architectures. *In Proceedings of the 6th International Workshop on High-Level Synthesis,* pages 16-26, November 1992.

[33] R. K. Gupta and G. De Micheli. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design and Test of Computers,* pages 29-41, September 1993.

[34] R. Gupta. Co-synthesis of Hardware and Software for Digital Embedded Systems. PhD thesis, Stanford University, December 1993.

[35] J. Henkel, R. Ernst, U. Holtmann, and T. Benner. Adaptation of Partitioning and High-Level Synthesis in Hardware/Software Co-Synthesis. *Proc. of the International Conference on CAD,* pages 96-100, 1994.

[36] J. L. Hennessy and D. A. Patterson. Computer Architectures: A Quantitative Approach. Morgan Kaufmann, 1996.

[37] C.Y. III Hitchcock. Addressing Modes for Fast and Optimal Code Generation. Phd thesis, Carnegie-Mellon University, December 1987.

[38] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225-230, December 1973.

[39] L.P. Horwitz, R.M. Karp, R.E. Miller, and S. Winograd. Index register allocation. *Journal of the ACM*, 13(1):43-61, January 1966.

[40] F. Irigoin and R. Triolet. Super-node partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[41] A. Kalavade and E. A. Lee. A Hardware-Software Codesign Methodology for DSP Applications. *IEEE Design and Test of Computers*, pages 16-28, September 1993.

[42] K. Keutzer. Personal communication to Stan Liao, 1995.

[43] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.

[44] M. S. Lam. An Effective Scheduling Technique for VLIW Machines. *In Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation,* pages 318-328, June 1988.

[45] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimization of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63-74, Santa Clara, California, April 1991

[46] D. Lamb. Construction of a Peephole Optimizer. *Software-Practices and Experiments*, 11(6):638-647, 1981.

[47] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. Kluwer Academic Publishers, Boston, MA, 1995.

[48] P. Lapsley, J. Bier, A. Shoham, and E. Lee. *DSP Processor Fundamentals- Architectures and Features.* IEEE Press, 1997.

[49] E. A. Lee. Programmable DSP Architectures: Part I. *IEEE ASSP Magazine*, pages 4-19, October 1988.

[50] E. A. Lee. Programmable DSP Architectures: Part II. *IEEE ASSP Magazine*, pages 4-14, January 1989.

[51] S. Liao. Code Generation and Optimization for Embedded Digital Signal Processors. PhD thesis, MIT Department of EECS, January 1996.

[52] S. Liao et al. Storage Assignment to Decrease Code Size. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 186–196, 1995. (This is a preliminary version of [53].)

[53] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[54] S. Liao, K. Keutzer, S. Tjiang, and S. Devadas. A new viewpoint on code generation for directed acyclic graphs. *ACM Transactions on Design Automation of Electronic Systems*, 3(1):51–75, January 1998.

[55] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Proceedings of International Conference on Computer-Aided Design*, pages 109-112, 1996.

[56] R. Leupers, A. Basu and P. Marwedel. Optimized Array Index Computation in DSP Programs. *ASP-DAC*, Yokohama, Japan, Feb 1998.

[57] R. Leupers and P. Marwedel. A Uniform Optimization Technique for Offset Assignment Problems. In *Proceedings of International Symposium on System Synthesis*, pages 3–8, 1998.

[58] C. Lieum, P. Paulin, and A. Jerraya. Address calculation for retargetable compilation and exploration of instruction-set architectures. In *Proceedings if the 33rd Design Automation Conference*, pages 597-600, June 1996.

[59] W. McKeeman. Peephole Optimization. *Communications of the ACM*, 8(7):443-444, 1965.

[60] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96-103, 1979.

[61] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[62] P.R. Panda. Memory Optimizations and Exploration for Embedded Systems. PhD thesis, UC Irvine Dept. of Information and Computer Science, 1998.

[63] P. G. Paulin, C. Lieum, T. C. May, and S. Sutarwala. DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective. *Journal of VLSI Signal Processing*, 9(1/2):23-47, January 1995.

[64] J. Ramanujam and P. Sadayappan. Nested loop tiling for distributed memory machines. In *Proceedings of the 5th Distributed Memory Computing Conference (DMCC5)*, pages 1088–1096, Charleston, SC, April 1990.

[65] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for non-shared memory machines. In *Proceedings Supercomputing 91,* pages 111-120, 1991.

[66] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems,* 2(4):472–482, October 1991.

[67] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing,* 16(2):108–120, October 1992.

[68] J. Ramanujam and P. Sadayappan. Iteration space tiling for distributed memory machines. In *Languages, Compilers and Environments for Distributed Memory Machines,* J. Saltz and P. Mehrotra, (Eds.), Amsterdam, The Netherlands: North-Holland, pages 255–270, 1992.

[69] J. Ramanujam, J. Hong, M. Kandemir, and S. Atri. Address register-oriented optimizations for embedded processors. In *Proc. 9th Workshop on Compilers for Parallel Computers (CPC 2001),* pp. 281–290, Edinburgh, Scotland, June 2001.

[70] A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded Dsps. *SIGPLAN '99, Atlanta, GA, USA,* pages 128-138, May 1999.

[71] K. L. Short, *Embedded Microprocessor Systems Design.* Prentice-Hall, 1998.

[72] A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *Proceedings of International Conference on Computer Aided Design*, pages 388-392, 1995.

[73] A. Sudarsanam, S. Liao and S. Devadas. Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 287–292, 1997.

[74] M.M. Strout, L. Carter, J. Ferrante and B. Simon. Schedule-Independent Storage Mappings for Loops. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA October 1998.

[75] D. E. Thomas, J. K. Adams, and H. Schmit. A Model and Methodology for Hardware-Software Codesign. *IEEE Design and Test of Computers,* pages 6-15, September 1993.

[76] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction Set Definition and Instruction Selection For ASIPs. In *Proceedings of the 7th IEEE/ACM International Symposium on High-Level Synthesis,* May 1994.

[77] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):31-44, April 1991.

[78] B. Wess. On the optimal code generation for signal flow computation. In *Proceedings of International Conference Circuits and Systems*, vol. 1, pages 444-447, 1990.

[79] Michael E. Wolf. Improving Locality and Parallelism in Nested Loops. PhD Thesis, Dept. of Computer Science, Stanford University, August 1992.

[80] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proc. 3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[81] Michael J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655-664, Reno, Nevada, November 1989.

[82] Michael J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[83] V. Zivojnovic, J. Velarde, and C. Schlager. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the 5th International Conference on Signal Processing Applications and Technology*, October 1994.

[84] Texas Instruments. TMS320C2x User's Guide, January 1993. Revision C.

VITA

Jinpyo Hong is from Taegu, Korea. After receiving a bachelor and a master of engineering degree in Computer Engineering from Kyungpook National University in 1992 and 1994 respectively, he worked for three and half years for KEPRI (Korea Electrical Power Research Institute). He joined the graduate program in Electrical and Computer Engineering at Louisiana State University in the Fall of 1997. He expects to receive his PhD degree in Electrical Engineering in August, 2002.